CHEF DOCUMENTS

Resources and Providers Reference

A resource is a statement of configuration policy. It describes the desired state of an element of your infrastructure, along with the steps needed to bring that item to the desired state. Each resource statement in a Chef recipe corresponds to a specific part of your infrastructure: a file, a template, a directory, a package, a service, a command to be executed, and so on. Each resource statement includes the resource type (such as `template`, `service` or `package`), its name, any attributes that specify additional details, and an action that tells the chef-client how to implement the configuration policy.

Together, resources describe all the components in your network. Recipes group resources together and describe working configurations. Add recipes to a run-list to describe the desired state for every node to which that run-list is assigned. Cookbooks are collections of recipes and are stored on the Chef server.

Where a resource represents a piece of the system (and its desired state), a provider defines the steps that are needed to bring that piece of the system from its current state into the desired state.

The `Chef::Platform` class maps providers to platforms (and platform versions). At the beginning of every chef-client run, Ohai verifies the `platform` and `platform_version` attributes on each node. The chef-client then uses those values to identify the correct provider, build an instance of that provider, identify the current state of the resource, do the specified action, and then mark the resource as updated (if changes were made).

For example:

```
directory "/tmp/folder" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

The chef-client will look up the provider for the `directory` resource, which happens to be `Chef::Provider::Directory`, call `load_current_resource` to create a `directory["/tmp/folder"]` resource, and then, based on the current state of the directory, do the specified action, which in this case is to create a directory called `/tmp/folder`. If the directory already exists, nothing will happen. If the directory was changed in any way, the resource is marked as updated.

This reference describes each of the resources available to the chef-client, including the list of actions available for the resource, the attributes that can be used, the providers that will do the work (and the provider's shortcut resource name), and examples of using each resource.

Common Functionality

The attributes and actions in this section apply to all resources.

Actions

The following actions are common to every resource:

Action	Description
`:nothing`	Use to define a resource that does nothing. This action is often used to define a resource that is later notified by other resources.

Examples

The following examples show how to use common actions in a recipe.

Use the :nothing action

```
service "memcached" do
  action :nothing
  supports :status => true, :start => true, :stop => true, :restart => true
end
```

Attributes

The following attributes are common to every resource:

Parameter	Description
`ignore_failure`	Use to continue running a recipe if a resource fails for any reason. Default value: `false`.

Parameter	Description
`provider`	Optional. The chef-client will attempt to determine the correct provider during the chef-client run, and then choose the best/correct provider based on configuration data collected at the start of the chef-client run. In general, a specific provider does not need to be specified.
`retries`	Use to specify the number of times to catch exceptions and retry the resource. Default value: `0`.
`retry_delay`	Use to specify the retry delay (in seconds). Default value: 2.
`sensitive`	Use to ensure that sensitive resource data is not logged by the chef-client. Default value: `false`.
`supports`	Use to specify a hash of options that contains hints about the capabilities of a resource. The chef-client may use these hints to help identify the correct provider. This attribute is only used by a small number of providers, including `User` and `Service`.

Provider

The chef-client will determine the correct provider based on configuration data collected by Ohai at the start of the chef-client run. This configuration data is then mapped to a platform and an associated list of providers.

Generally, it's best to let the chef-client choose the provider and this is (by far) the most common approach. However, in some cases specifying a provider may be desirable. There are two approaches:

- Use a more specific short name—`yum_package "foo" do` instead of `package "foo" do`, `script "foo" do` instead of `bash "foo" do`, and so on—when available
- Use the `provider` attribute to specify the long name as an attribute of a resource, e.g. `provider Chef::Provider::Long::Name`

Examples

The following examples show how to use common attributes in a recipe.

Use the ignore_failure common attribute

```
gem_package "syntax" do
  action :install
  ignore_failure true
end
```

Use the provider common attribute

```
package "some_package" do
  provider Chef::Provider::Package::Rubygems
end
```

Use the supports common attribute

```
service "apache" do
  supports :restart => true, :reload => true
  action :enable
end
```

Use the supports and providers common attributes

```
service "some_service" do
  provider Chef::Provider::Service::Upstart
  supports :status => true, :restart => true, :reload => true
  action [ :enable, :start ]
end
```

Guards

A guard attribute can be used to evaluate the state of a node during the execution phase of the chef-client run. Based on the results of this evaluation, a guard attribute is then used to tell the chef-client if it should continue executing a resource. A guard attribute accepts either a string value or a Ruby block value:

- A string is executed as a shell command. If the command returns `0`, the guard is applied. If the command returns any other value, then the guard attribute is not applied.
- A block is executed as Ruby code that must return either `true` or `false`. If the block returns `true`, the guard attribute is applied. If the block returns `false`, the guard attribute is not applied.

A guard attribute is useful for ensuring that a resource is idempotent by allowing that resource to test for the desired state as it is being executed, and then if the desired state is present, for the chef-client to do nothing.

Attributes

The following attributes can be used to define a guard that is evaluated during the execution phase of the chef-client run:

Guard	Description
`not_if`	Use to prevent a resource from executing when the condition returns `true`.
`only_if`	Use to allow a resource to execute only if the condition returns `true`.

Arguments

The following arguments can be used with the `not_if` or `only_if` guard attributes:

Argument	Description
`:user`	Use to specify the user that a command will run as. For example: `not_if "grep adam /etc/passwd", :user => 'adam'`
`:group`	Use to specify the group that a command will run as. For example: `not_if "grep adam /etc/passwd", :group => 'adam'`
`:environment`	Use to specify a Hash of environment variables to be set. For example: `not_if "grep adam /etc/passwd", :environment => { 'HOME' => "/home/adam" }`
`:cwd`	Use to set the current working directory before running a command. For example: `not_if "grep adam passwd", :cwd => '/etc'`
`:timeout`	Use to set a timeout for a command. For example: `not_if "sleep 10000", :timeout => 10`

not_if Examples

The following examples show how to use `not_if` as a condition in a recipe:

Create a file, but not if an attribute has a specific value

The following example shows how to use the `not_if` condition to create a file based on a template and using the presence of an attribute on the node to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if { node[:some_value] }
end
```

Create a file with a Ruby block, but not if "/etc/passwd" exists

The following example shows how to use the `not_if` condition to create a file based on a template and then Ruby code to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if do
    File.exists?("/etc/passwd")
  end
end
```

Create a file with Ruby block that has curly braces, but not if "/etc/passwd" exists

The following example shows how to use the `not_if` condition to create a file based on a template and using a Ruby block (with curly braces) to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if {File.exists?("/etc/passwd")}
end
```

Create a file using a string, but not if "/etc/passwd" exists

The following example shows how to use the `not_if` condition to create a file based on a template and using a string to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if "test -f /etc/passwd"
end
```

Install a file from a remote location using bash

The following is an example of how to install the `foo123` module for Nginx. This module adds shell-style functionality to an Nginx configuration file and does the following:

- Declares three variables
- Gets the Nginx file from a remote location
- Installs the file using Bash to the path specified by the `src_filepath` variable

```
#  the following code sample is similar to the ``upload_progress_module`` recipe in the ``nginx`` cookbook:

src_filename = "foo123-nginx-module-v#{node['nginx']['foo123']['version']}.tar.gz"
src_filepath = "#{Chef::Config['file_cache_path']}/#{src_filename}"
extract_path = "#{Chef::Config['file_cache_path']}/nginx_foo123_module/#{node['nginx']['foo123']['checksum']

remote_file src_filepath do
  source node['nginx']['foo123']['url']
  checksum node['nginx']['foo123']['checksum']
  owner 'root'
  group 'root'
  mode '0644'
end

bash 'extract_module' do
  cwd ::File.dirname(src_filepath)
  code <<-EOH
    mkdir -p #{extract_path}
    tar xzf #{src_filename} -C #{extract_path}
    mv #{extract_path}/*/* #{extract_path}/
    EOH
  not_if { ::File.exists?(extract_path) }
end
```

only_if Examples

The following examples show how to use `only_if` as a condition in a recipe:

Create a file, but only if an attribute has a specific value

The following example shows how to use the `only_if` condition to create a file based on a template and using the presence of an attribute on the node to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  only_if { node[:some_value] }
end
```

Create a file with a Ruby block, but only if "/etc/passwd" does not exist

The following example shows how to use the `only_if` condition to create a file based on a template, and then use Ruby to specify a condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  only_if do ! File.exists?("/etc/passwd") end
end
```

Create a file using a string, but only if "/etc/passwd" exists

The following example shows how to use the `only_if` condition to create a file based on a template and using a string to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  only_if "test -f /etc/passwd"
end
```

Guard Interpreters

Any resource that passes a string command may also specify the interpreter that will be used to evaluate that string command. This is done by using the `guard_interpreter` attribute to specify a **script**-based resource.

Attributes

The `guard_interpreter` attribute may be set to any of the following values:

Value	Description
`:bash`	Use to evaluate a string command using the **bash** resource.
`:batch`	Use to evaluate a string command using the **batch** resource.
`:csh`	Use to evaluate a string command using the **csh** resource.
`:default`	Default. Use to execute the default interpreter as identified by the chef-client.
`:perl`	Use to evaluate a string command using the **perl** resource.
`:powershell_script`	Use to evaluate a string command using the **powershell_script** resource.
`:python`	Use to evaluate a string command using the **python** resource.
`:ruby`	Use to evaluate a string command using the **ruby** resource.

Inheritance

All non-default interpreters will **not** inherit arguments that are available to guard attributes unless the `guard_interpreter` attribute is specified. For example, the following resource block will not inherit the `environment` attribute (and requires that the environment variable be specified within the `not_if` guard in addition to the resource block itself):

```
bash "javatooling" do
  environment {"JAVA_HOME" => "/usr/lib/java/jdk1.7/home"}
  code "java-based-daemon-ctl.sh -start"
  not_if "java-based-daemon-ctl.sh -test-started, :environment {'JAVA_HOME' => '/usr/lib/java/jdk1.7/home'}"
end
```

and the following resource block will inherit the `environment` attribute:

```
bash "javatooling" do
  guard_interpreter :bash
  environment {"JAVA_HOME" => "/usr/lib/java/jdk1.7/home"}
  code "java-based-daemon-ctl.sh -start"
  not_if "java-based-daemon-ctl.sh -test-started"
end
```

Examples

For example, the following code block will ensure the command is evaluated using the default intepreter as identified by the chef-client:

```
resource #name do
  guard_interpreter :default
  # code
end
```

Lazy Attribute Evaluation

In some cases, the value for an attribute cannot be known until the execution phase of a chef-client run. In this situation, using lazy evaluation of attribute values can be helpful. Instead of an attribute being assigned a value, it may instead be assigned a code block. The syntax for using lazy evaluation is as follows:

```
attribute_name lazy { code_block }
```

where `lazy` is used to tell the chef-client to evaluate the contents of the code block later on in the resource evaluation process (instead of immediately) and `{ code_block }` is arbitrary Ruby code that provides the value.

For example, a resource that is not doing lazy evaluation:

```
template "template_name" do
  # some attributes
  path "/foo/bar"
end
```

and a resource that is doing lazy evaluation:

```
template "template_name" do
  # some attributes
  path lazy { " some Ruby code " }
end
```

In the previous examples, the first resource uses the value `/foo/bar` and the second resource uses the value provided by the code block, as long as the contents of that code block are a valid resource attribute.

Notifications

The following notifications can be used with any resource:

Notification	Description
`notifies`	Use to notify another resource to take an action if this resource's state changes for any reason.
`subscribes`	Use to take action on this resource if another resource's state changes. This is similar to `notifies`, but reversed.

Notifications Timers

The following timers can be used to define when a notification is triggered:

Timer	Description
`:delayed`	Use to specify that a notification should be queued up and then executed at the very end of a chef-client run.
`:immediately`	Use to specify that a notification be run immediately.

Notifies Syntax

The basic syntax of a `notifies` notification is:

```
resource "name" do
  notifies :notification, "resource[name]", :timer
end
```

Examples

The following examples show how to use the `notifies` notification in a recipe.

Delay notifications

```
template "/etc/nagios3/configures-nagios.conf" do
  # other parameters
  notifies :run, "execute[test-nagios-config]", :delayed
end
```

Notify immediately

By default, notifications are `:delayed`, that is they are queued up as they are triggered, and then executed at the very end of a chef-client run. To run an action immediately, use `:immediately`:

```
template "/etc/nagios3/configures-nagios.conf" do
  # other parameters
  notifies :run, "execute[test-nagios-config]", :immediately
end
```

and then the chef-client would immediately run the following:

```
execute "test-nagios-config" do
  command "nagios3 --verify-config"
  action :nothing
end
```

Enable a service after a restart or reload

```
service "apache" do
  supports :restart => true, :reload => true
  action :enable
end
```

Notify multiple resources

```
template "/etc/chef/server.rb" do
  source "server.rb.erb"
  owner 'root'
  group 'root'
  mode '0644'
  notifies :restart, "service[chef-solr]", :delayed
  notifies :restart, "service[chef-solr-indexer]", :delayed
  notifies :restart, "service[chef-server]", :delayed
end
```

Notify in a specific order

To notify multiple resources, and then have these resources run in a certain order, do something like the following:

```
execute 'foo' do
  command '...'
  notifies :run, 'template[baz]', :immediately
  notifies :install, 'package[bar]', :immediately
  notifies :run, 'execute[final]', :immediately
end

template 'baz' do
  ...
  notifies :run, 'execute[restart_baz]', :immediately
end

package 'bar'

execute 'restart_baz'

execute 'final' do
  command '...'
end
```

where the sequencing will be in the same order as the resources are listed in the recipe: `execute 'foo'`, `template 'baz'`, `execute [restart_baz]`, `package 'bar'`, and `execute 'final'`.

Reload a service

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  notifies :reload, "service[apache]", :immediately
end
```

Restart a service when a template is modified

```
template "/etc/www/configures-apache.conf" do
  notifies :restart, "service[apache]", :immediately
end
```

Send notifications to multiple resources

To send notifications to multiple resources, just use multiple attributes. Multiple attributes will get sent to the notified resources in the order specified.

```
template "/etc/netatalk/netatalk.conf" do
  notifies :restart, "service[afpd]", :immediately
  notifies :restart, "service[cnid]", :immediately
end

service "afpd"
service "cnid"
```

Execute a command using a template

The following example shows how to set up IPv4 packet forwarding using the **execute** resource to run a command named `forward_ipv4` that uses a template defined by the **template** resource:

```
execute "forward_ipv4" do
  command "echo > /proc/.../ipv4/ip_forward"
  action :nothing
end

template "/etc/file_name.conf" do
  source "routing/file_name.conf.erb"
  notifies :run, 'execute[forward_ipv4]', :delayed
end
```

where the `command` attribute for the **execute** resource contains the command that is to be run and the `source` attribute for the **template** resource specifies which template to use. The `notifies` attribute for the **template** specifies that the `execute[forward_ipv4]` (which is defined by the **execute** resource) should be queued up and run at the end of the chef-client run.

Restart a service, and then notify a different service

The following example shows how start a service named `example_service` and immediately notify the Nginx service to restart.

```
service "example_service" do
  action :start
  provider Chef::Provider::Service::Init
  notifies :restart, "service[nginx]", :immediately
end
```

where by using the default `provider` for the **service**, the recipe is telling the chef-client to determine the specific provider to be used during the chef-client run based on the platform of the node on which the recipe will run.

Notify when a remote source changes

```
remote_file "/tmp/couch.png" do
  source "http://couchdb.apache.org/img/sketch.png"
  action :nothing
end

http_request "HEAD http://couchdb.apache.org/img/sketch.png" do
  message ""
  url "http://couchdb.apache.org/img/sketch.png"
  action :head
  if File.exists?("/tmp/couch.png")
    headers "If-Modified-Since" => File.mtime("/tmp/couch.png").httpdate
  end
  notifies :create, "remote_file[/tmp/couch.png]", :immediately
end
```

Subscribes Syntax

The basic syntax of a `subscribes` notification is:

```
resource "name" do
  subscribes :notification, "resource[name]", :timer
end
```

Examples

The following examples show how to use the `subscribes` notification in a recipe.

Prevent restart and reconfigure if configuration is broken

Use the `:nothing` common action to prevent an application from restarting, and then use the `subscribes` notification to ask the broken configuration to be reconfigured immediately:

```
execute "test-nagios-config" do
  command "nagios3 --verify-config"
  action :nothing
  subscribes :run, "template[/etc/nagios3/configures-nagios.conf]", :immediately
end
```

Reload a service using a template

To reload a service based on a template, use the **template** and **service** resources together in the same recipe, similar to the following:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
end

service "apache" do
  supports :restart => true, :reload => true
  action :enable
  subscribes :reload, "template[/tmp/somefile]", :immediately
end
```

where the `subscribes` notification is used to reload the service using the template specified by the **template** resource.

Stash a file in a data bag

The following example shows how to use the **ruby_block** resource to stash a BitTorrent file in a data bag so that it can be distributed to nodes in the organization.

```
#  the following code sample comes from the ``seed`` recipe in the following cookbook: https://github.com/ma

ruby_block "share the torrent file" do
  block do
    f = File.open(node['bittorrent']['torrent'],'rb')
    #read the .torrent file and base64 encode it
    enc = Base64.encode64(f.read)
    data = {
      'id'=>bittorrent_item_id(node['bittorrent']['file']),
      'seed'=>node.ipaddress,
      'torrent'=>enc
    }
    item = Chef::DataBagItem.new
    item.data_bag('bittorrent')
    item.raw_data = data
    item.save
  end
  action :nothing
  subscribes :create, "bittorrent_torrent[#{node['bittorrent']['torrent']}]", :immediately
end
```

Relative Paths

The following relative paths can be used with any resource:

Relative Path	Description
`#{ENV['HOME']}`	Use to return the ~ path in Linux and Mac OS X or the `%HOMEPATH%` in Microsoft Windows.

Examples

```
template "#{ENV['HOME']}/chef-getting-started.txt" do
  source "chef-getting-started.txt.erb"
  mode '0644'
end
```

Run Resources from the Resource Collection

The chef-client processes recipes in two phases:

1. First, each resource in the node object is identified and a resource collection is built. All recipes are loaded in a specific order, and then the actions specified within each of them are identified.
2. Next, the chef-client configures the system based on the order of the resources in the resource collection. Each resource is mapped to a provider, which then examines the node and then does the steps necessary to complete the action.

Sometimes, it may be necessary to ensure that a specific resource is run during the phase that builds the resource collection. For example:

- A resource may need to run first so that it can download a package that will be used by other resources in the resource collection
- Several resources need to install a package; rather than having the package installer run several times, it can be configured to run only once

To support these types of uses cases, it is possible to tell the chef-client to run a resource at the beginning and/or the end of the resource collection phase. Effectively, run a resource before all other resources are added to the resource collection and/or after all resources have been added, but before the chef-client configures the system.

Before other resources

To run a resource at the start of the resource collection phase of the chef-client run, set up a `Chef::Resource` object, and then call the method that runs the action.

Update a package cache

It is important to make sure that an operating system's package cache is up to date before installing packages, otherwise there may be references to versions that no longer exist. For example, on Debian or Ubuntu systems, the Apt cache needs to be updated. Use code similar to the following:

```
e = execute "apt-get update" do
  action :nothing
end

e.run_action(:run)
```

where `e` is created as a `Chef::Resource::Execute` Ruby object. The `action` attribute is set to `:nothing` so that the `run_action` method can be used to tell the chef-client to run the specified command. The **apt** (for Debian and Ubuntu) and **pacman** (for Arch Linux) cookbooks can be used for this purpose. The preceding recipe can be placed at the top of a node's run list to ensure it is run before the chef-client tries to install any packages.

An anti-pattern

Unfortunately, resources that are executed when the resource collection is being built cannot notify any resource that has yet to be added to the resource collection. For example:

```
execute "ifconfig"

p = package 'vim-enhanced' do
  action :nothing
  notifies :run, "execute[ifconfig]", :immediately
end
p.run_action(:install)
```

In some cases, the better approach may be to install the package before the resource collection is built to ensure that it is available to other resources later on. Or, something like the following can be used:

```
p = package "foo" do
```

```
  #parameters
end
p.run_action(:install)

if p.updated_by_last_action?
  #Call the resource that we want to "notify"
end
```

After the resource collection is built

To run a resource at the end of the resource collection phase of the chef-client run, use the `:delayed` timer on a notification.

Atomic File Updates

Atomic updates are used with **file**-based resources to help ensure that file updates can be made when updating a binary or if disk space runs out.

Atomic updates are enabled by default. They can be managed globally using the `file_atomic_update` attribute in the client.rb file. They can be managed on a per-resource basis using the `atomic_update` attribute that is available with the **cookbook_file**, **file**, **remote_file**, and **template** resources.

Note

On certain platforms, and after a file has been moved into place, the chef-client may modify file permissions to support features specific to those platforms. On platforms with SELinux enabled, the chef-client will fix up the security contexts after a file has been moved into the correct location by running the `restorecon` command. On the Microsoft Windows platform, the chef-client will create files so that ACL inheritance works as expected.

Windows File Security

To support Microsoft Windows security, the **template**, **file**, **remote_file**, **cookbook_file**, **directory**, and **remote_directory** resources support the use of inheritance and access control lists (ACLs) within recipes.

Access Control Lists (ACLs)

The `rights` attribute can be used in a recipe to manage access control lists (ACLs), which allow permissions to be given to multiple users and groups. The syntax for the `rights` attribute is as follows:

```
rights permission, principal, option_type => value
```

where

- `permission` is used to specify which rights will be granted to the `principal`. The possible values are: `:read`, `:write`, `read_execute`, `:modify`, `:full_control`, and `:deny`.

  These permissions are cumulative. If `:write` is specified, then it includes `:read`. If `:full_control` is specified, then it includes both `:write` and `:read`. If `:deny` is specified, then the user or group will not have rights to the object.

  (For those who know the Microsoft Windows API: `:read` corresponds to `GENERIC_READ`; `:write` corresponds to `GENERIC_WRITE`; `:read_execute` corresponds to `GENERIC_READ` and `GENERIC_EXECUTE`; `:modify` corresponds to `GENERIC_WRITE`, `GENERIC_READ`, `GENERIC_EXECUTE`, and `DELETE`; `:full_control` corresponds to `GENERIC_ALL`, which allows a user to change the owner and other metadata about a file.)

- `principal` is used to specify a group or user name. This is identical to what is entered in the login box for Microsoft Windows, such as `user_name`, `domain\user_name`, or `user_name@fully_qualified_domain_name`. The chef-client does not need to know if a principal is a user or a group.
- `option_type` is a hash that contains advanced rights options. For example, the rights to a directory that only applies to the first level of children might look something like: `rights :write, "domain\group_name", :one_level_deep => true`. Possible option types:

Option Type	Description
`:applies_to_children`	Use to specify how permissions are applied to children. Possible values: `true` to inherit both child directories and files; `false` to not inherit any child directories or files; `:containers_only` to inherit only child directories (and not files); `:objects_only` to recursively inherit files (and not child directories).
`:applies_to_self`	Indicates whether a permission is applied to the parent directory. Possible values: `true` to apply to the parent directory or file and its children; `false` to not apply only to child directories and files.
`:one_level_deep`	Indicates the depth to which permissions will be applied. Possible values: `true` to apply only to the first level of children; `false` to apply to all children.

The `rights` attribute can be used as many times as necessary; the chef-client will apply them to the file or directory as required. For example:

```
resource "x.txt" do
  rights :read, "Everyone"
  rights :write, "domain\group"
  rights :full_control, "group_name_or_user_name"
  rights :full_control, "user_name", :applies_to_children => true
end
```

or:

```
rights :read, ["Administrators","Everyone"]
rights :deny, ["Julian", "Lewis"]
rights :full_control, "Users", :applies_to_children => true
rights :write, "Sally", :applies_to_children => :containers_only, :applies_to_self => false, :one_level_deep
```

Some other important things to know when using the `rights` attribute:

- Order independence. It doesn't matter if `rights :deny, ["Julian", "Lewis"]` is placed before or after `rights :read, ["Julian", "Lewis"]`, both Julian and Lewis will be unable to read the document.
- Only inherited rights remain. All existing explicit rights on the object are removed and replaced.
- If rights are not specified, nothing will be changed. The chef-client does not clear out the rights on a file or directory if rights are not specified.
- Changing inherited rights can be expensive. Microsoft Windows will propagate rights to all children recursively due to inheritance. This is a normal aspect of Microsoft Windows, so consider the frequency with which this type of action is necessary and take steps to control this type of action if performance is the primary consideration.

Inheritance

By default, a file or directory inherits rights from its parent directory. Most of the time this is the preferred behavior, but sometimes it may be necessary to take steps to more specifically control rights. The `inherits` attribute can be used to specifically tell the chef-client to apply (or not apply) inherited rights from its parent directory.

For example, the following example specifies the rights for a directory:

```
directory 'C:\mordor' do
  rights :read, 'MORDOR\Minions'
  rights :full_control, 'MORDOR\Sauron'
end
```

and then the following example specifies how to use inheritance to deny access to the child directory:

```
directory 'C:\mordor\mount_doom' do
  rights :full_control, 'MORDOR\Sauron'
  inherits false # Sauron is the only person who should have any sort of access
end
```

If the `:deny` permission were to be used instead, something could slip through unless all users and groups were denied.

Another example also shows how to specify rights for a directory:

```
directory 'C:\mordor' do
  rights :read, 'MORDOR\Minions'
  rights :full_control, 'MORDOR\Sauron'
  rights :write, 'SHIRE\Frodo' # Who put that there I didn't put that there
end
```

but then not use the `inherits` attribute to deny those rights on a child directory:

```
directory 'C:\mordor\mount_doom' do
  rights :deny, 'MORDOR\Minions' # Oops, not specific enough
end
```

Because the `inherits` attribute is not specified, the chef-client will default it to `true`, which will ensure that security settings for existing files remain unchanged.

Resources

The following resources are platform resources with built-in providers:

- apt_package (based on the package resource)
- bash
- batch
- breakpoint
- chef_gem (based on the package resource)
- chef_handler (available from the chef_handler cookbook)
- cookbook_file
- cron

- csh
- deploy (including git and Subversion)
- directory
- dpkg_package (based on the package resource)
- dsc_script
- easy_install_package (based on the package resource)
- env
- erl_call
- execute
- file
- freebsd_package (based on the package resource)
- gem_package (based on the package resource)
- git
- group
- http_request
- ifconfig
- ips_package (based on the package resource)
- link
- log
- macports_package (based on the package resource)
- mdadm
- mount
- ohai
- package
- pacman_package (based on the package resource)
- perl
- portage_package (based on the package resource)
- powershell_script
- python
- registry_key
- remote_directory
- remote_file
- route
- rpm_package (based on the package resource)
- ruby
- ruby_block
- script
- service
- smartos_package (based on the package resource)
- solaris_package (based on the package resource)
- subversion
- template
- user
- windows_package
- yum (based on the package resource)

See below for more information about each of these resources, their related actions and attributes, the providers they rely on, and examples of how these resources can be used in recipes.

apt_package

Use the **apt_package** resource to manage packages for the Debian and Ubuntu platforms.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **apt_package** resource in a recipe is as follows:

```
apt_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `apt_package` tells the chef-client to use the `Chef::Provider::Package::Apt` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:reconfig`	Use to reconfigure a package. This action requires a response file.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`arch`	The architecture of the package that will be installed or upgraded. (This value can also be passed as part of the package name.)
`options`	One (or more) additional options that are passed to the command. For example, common apt-get directives, such as `--no-install-recommends`. See the apt-get man page for the full list.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The direct path to a dpkg or deb package.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Apt`	`apt_package`	The provider that is used with the Debian and Ubuntu platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package using package manager

```
apt_package "name of package" do
  action :install
end
```

Install a package using local file

```
apt_package "jwhois" do
  action :install
  source '/path/to/jwhois.deb'
end
```

Install without using recommend packages as a dependency

```
package "apache2" do
  options "--no-install-recommends"
end
```

bash

Use the **bash** resource to execute scripts using the Bash interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **bash** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **bash** resource in a recipe is as follows:

```
bash "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `bash` tells the chef-client to use the `Chef::Resource::Script::Bash` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.

Attribute	Description
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Bash`	`bash`	The provider that is used with the Bash command interpreter.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Use a named provider to run a script

```
bash "install_something" do
  user "root"
  cwd "/tmp"
  code <<-EOH
  wget http://www.example.com/tarball.tar.gz
  tar -zxf tarball.tar.gz
  cd tarball
  ./configure
  make
  make install
  EOH
end
```

Install a file from a remote location using bash

The following is an example of how to install the `foo123` module for Nginx. This module adds shell-style functionality to an Nginx configuration file and does the following:

- Declares three variables
- Gets the Nginx file from a remote location
- Installs the file using Bash to the path specified by the `src_filepath` variable

```
#  the following code sample is similar to the ``upload_progress_module`` recipe in the ``nginx`` cookbook:

src_filename = "foo123-nginx-module-v#{node['nginx']['foo123']['version']}.tar.gz"
src_filepath = "#{Chef::Config['file_cache_path']}/#{src_filename}"
extract_path = "#{Chef::Config['file_cache_path']}/nginx_foo123_module/#{node['nginx']['foo123']['checksum']

remote_file src_filepath do
  source node['nginx']['foo123']['url']
  checksum node['nginx']['foo123']['checksum']
  owner 'root'
  group 'root'
  mode '0644'
end

bash 'extract_module' do
  cwd ::File.dirname(src_filepath)
  code <<-EOH
    mkdir -p #{extract_path}
    tar xzf #{src_filename} -C #{extract_path}
    mv #{extract_path}/*/* #{extract_path}/
    EOH
  not_if { ::File.exists?(extract_path) }
end
```

Install an application from git using bash

The following example shows how Bash can be used to install a plug-in for rbenv named `ruby-build`, which is located in git version source control. First, the application is synchronized, and then Bash changes its working directory to the location in which `ruby-build` is located, and then runs a command.

```
 git "#{Chef::Config[:file_cache_path]}/ruby-build" do
   repository "git://github.com/sstephenson/ruby-build.git"
```

```
    reference "master"
    action :sync
  end

  bash "install_ruby_build" do
    cwd "#{Chef::Config[:file_cache_path]}/ruby-build"
    user "rbenv"
    group "rbenv"
    code <<-EOH
      ./install.sh
      EOH
    environment 'PREFIX' => "/usr/local"
end
```

To read more about `ruby-build`, see here: https://github.com/sstephenson/ruby-build.

Store certain settings

The following recipe shows how an attributes file can be used to store certain settings. An attributes file is located in the `attributes/` directory in the same cookbook as the recipe which calls the attributes file. In this example, the attributes file specifies certain settings for Python that are then used across all nodes against which this recipe will run.

Python packages have versions, installation directories, URLs, and checksum files. An attributes file that exists to support this type of recipe would include settings like the following:

```
default['python']['version'] = '2.7.1'

if python['install_method'] == 'package'
  default['python']['prefix_dir'] = '/usr'
else
  default['python']['prefix_dir'] = '/usr/local'
end

default['python']['url'] = 'http://www.python.org/ftp/python'
default['python']['checksum'] = '80e387...85fd61'
```

and then the methods in the recipe may refer to these values. A recipe that is used to install Python will need to do the following:

- Identify each package to be installed (implied in this example, not shown)
- Define variables for the package `version` and the `install_path`
- Get the package from a remote location, but only if the package does not already exist on the target system
- Use the **bash** resource to install the package on the node, but only when the package is not already installed

```
#  the following code sample comes from the ``oc-nginx`` cookbook on |github|: https://github.com/cookbooks/

version = node['python']['version']
install_path = "#{node['python']['prefix_dir']}/lib/python#{version.split(/(^\d+\.\d+)/)[1]}"

remote_file "#{Chef::Config[:file_cache_path]}/Python-#{version}.tar.bz2" do
  source "#{node['python']['url']}/#{version}/Python-#{version}.tar.bz2"
  checksum node['python']['checksum']
  mode '0644'
  not_if { ::File.exists?(install_path) }
end

bash "build-and-install-python" do
  cwd Chef::Config[:file_cache_path]
  code <<-EOF
    tar -jxvf Python-#{version}.tar.bz2
    (cd Python-#{version} && ./configure #{configure_options})
    (cd Python-#{version} && make && make install)
  EOF
  not_if { ::File.exists?(install_path) }
end
```

batch

A resource defines the desired state for a single configuration item present on a node that is under management by Chef. A resource collection—one (or more) individual resources—defines the desired state for the entire node. During every chef-client run, the current state of each resource is tested, after which the chef-client will take any steps that are necessary to repair the node and bring it back into the desired state.

Use the **batch** resource to execute a batch script using the cmd.exe interpreter. The **batch** resource creates and executes a temporary file (similar to how the **script** resource behaves), rather than running the command inline. This resource inherits actions (`:run` and `:nothing`) and attributes (`creates`, `cwd`, `environment`, `group`, `path`, `timeout`, and `user`) from the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Syntax

The syntax for using the **batch** resource in a recipe is as follows:

```
batch "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `batch` tells the chef-client to use the `Chef::Provider::Batch` provider during the chef-client run
- `"name"` is the name of the batch script or the path to a file which contains it
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
batch "echo vars" do
  code "echo %TEMP% %SYSTEMDRIVE% %PATH% %WINDIR%"
  action :run
end
```

Actions

This resource has the following actions:

Action	Description
`:run`	Use to run a batch file.

Attributes

This resource has the following attributes:

Attribute	Description
`architecture`	The architecture of the process under which a script is executed. Possible values: `:x86` (for 32-bit processes) and `:x86_64` (for 64-bit processes). If these values are not provided in a recipe, the chef-client will default to the correct value for the architecture, as determined by Ohai. An exception will be raised when anything other than `:x86` is specified for a 32-bit process.
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed.
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory from which a command is run.
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	A user name or identifier that must be changed before running a command.

Note

See http://technet.microsoft.com/en-us/library/bb490880.aspx for more information about the cmd.exe interpreter.

Guards

A guard attribute can be used to evaluate the state of a node during the execution phase of the chef-client run. Based on the results of this evaluation, a guard attribute is then used to tell the chef-client if it should continue executing a resource. A guard attribute accepts either a string value or a Ruby block value:

- A string is executed as a shell command. If the command returns `0`, the guard is applied. If the command returns any other value, then the guard attribute is not applied.
- A block is executed as Ruby code that must return either `true` or `false`. If the block returns `true`, the guard attribute is applied. If the block returns `false`, the guard attribute is not applied.

A guard attribute is useful for ensuring that a resource is idempotent by allowing that resource to test for the desired state as it is being executed, and then if the desired state is present, for the chef-client to do nothing.

Attributes

The following attributes can be used to define a guard that is evaluated during the execution phase of the chef-client run:

Guard	Description
`not_if`	Use to prevent a resource from executing when the condition returns `true`.
`only_if`	Use to allow a resource to execute only if the condition returns `true`.

Arguments

The following arguments can be used with the `not_if` or `only_if` guard attributes:

Argument	Description
`:user`	Use to specify the user that a command will run as. For example: `not_if "grep adam /etc/passwd", :user => 'adam'`
`:group`	Use to specify the group that a command will run as. For example: `not_if "grep adam /etc/passwd", :group => 'adam'`
`:environment`	Use to specify a Hash of environment variables to be set. For example: `not_if "grep adam /etc/passwd", :environment => { 'HOME' => "/home/adam" }`
`:cwd`	Use to set the current working directory before running a command. For example: `not_if "grep adam passwd", :cwd => '/etc'`
`:timeout`	Use to set a timeout for a command. For example: `not_if "sleep 10000", :timeout => 10`

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Batch`	`batch`	The default provider for the Microsoft Windows platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Unzip a file, and then move it

To run a batch file that unzips and then moves Ruby, do something like:

```
batch "unzip_and_move_ruby" do
  code <<-EOH
    7z.exe x #{Chef::Config[:file_cache_path]}/ruby-1.8.7-p352-i386-mingw32.7z
      -oC:\\source -r -y
    xcopy C:\\source\\ruby-1.8.7-p352-i386-mingw32 C:\\ruby /e /y
    EOH
end

batch "echo some env vars" do
  code <<-EOH
    echo %TEMP%
    echo %SYSTEMDRIVE%
    echo %PATH%
    echo %WINDIR%
    EOH
end
```

or:

```
batch "unzip_and_move_ruby" do
```

```
  code <<-EOH
    7z.exe x #{Chef::Config[:file_cache_path]}/ruby-1.8.7-p352-i386-mingw32.7z
      -oC:\\source -r -y
    xcopy C:\\source\\ruby-1.8.7-p352-i386-mingw32 C:\\ruby /e /y
    EOH
end

batch "echo some env vars" do
  code "echo %TEMP%\\necho %SYSTEMDRIVE%\\necho %PATH%\\necho %WINDIR%"
end
```

breakpoint

A resource defines the desired state for a single configuration item present on a node that is under management by Chef. A resource collection—one (or more) individual resources—defines the desired state for the entire node. During every chef-client run, the current state of each resource is tested, after which the chef-client will take any steps that are necessary to repair the node and bring it back into the desired state.

Use the **breakpoint** resource to add breakpoints to recipes. Run the chef-client in chef-shell mode, and then use those breakpoints to debug recipes. Breakpoints are ignored by the chef-client during an actual chef-client run. That said, breakpoints are typically used to debug recipes only when running them in a non-production environment, after which they are removed from those recipes before the parent cookbook is uploaded to the Chef server.

Syntax

The syntax for using the **breakpoint** resource in a recipe is as follows:

```
breakpoint "name" do
  action :break
end
```

where

- `:break` will tell the chef-client to stop running a recipe; can only be used when the chef-client is being run in chef-shell mode

Actions

This resource has the following actions:

Action	Description
`:break`	Use to add a breakpoint to a recipe.

Attributes

This resource does not have any attributes.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Breakpoint`	breakpoint	The default provider for all recipes.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

A recipe without a breakpoint

```
yum_key node['yum']['elrepo']['key'] do
  url  node['yum']['elrepo']['key_url']
  action :add
end

yum_repository "elrepo" do
  description "ELRepo.org Community Enterprise Linux Extras Repository"
  key node['yum']['elrepo']['key']
  mirrorlist node['yum']['elrepo']['url']
  includepkgs node['yum']['elrepo']['includepkgs']
  exclude node['yum']['elrepo']['exclude']
  action :create
end
```

The same recipe with breakpoints

```
breakpoint "before yum_key node['yum']['repo_name']['key']" do
  action :break
end

yum_key node['yum']['repo_name']['key'] do
  url  node['yum']['repo_name']['key_url']
  action :add
end

breakpoint "after yum_key node['yum']['repo_name']['key']" do
  action :break
end

breakpoint "before yum_repository 'repo_name'" do
  action :break
end

yum_repository "repo_name" do
  description "description"
  key node['yum']['repo_name']['key']
  mirrorlist node['yum']['repo_name']['url']
  includepkgs node['yum']['repo_name']['includepkgs']
  exclude node['yum']['repo_name']['exclude']
  action :create
end

breakpoint "after yum_repository 'repo_name'" do
  action :break
end
```

where the `"name"` of each breakpoint is an arbitrary string. In the previous example, `"name"` is used to indicate if the breakpoint is before or after a resource, and then also to specify which resource.

chef_gem

Use the **chef_gem** resource to install a gem only for the instance of Ruby that is dedicated to the chef-client. When a package is installed from a local file, it must be added to the node using the **remote_file** or **cookbook_file** resources.

The **chef_gem** resource works with all of the same attributes and options as the **gem_package** resource, but does not accept the `gem_binary` attribute because it always uses the `CurrentGemEnvironment` under which the chef-client is running. In addition to performing actions similar to the **gem_package** resource, the **chef_gem** resource does the following:

- Runs its actions immediately, before convergence, allowing a gem to be used in a recipe immediately after it is installed
- Runs `Gem.clear_paths` after the action, ensuring that gem is aware of changes so that it can be required immediately after it is installed

Warning

The **chef_gem** and **gem_package** resources are both used to install Ruby gems. For any machine on which the chef-client is installed, there are two instances of Ruby. One is the standard, system-wide instance of Ruby and the other is a dedicated instance that is available only to the chef-client. Use the **chef_gem** resource to install gems into the instance of Ruby that is dedicated to the chef-client. Use the **gem_package** resource to install all other gems (i.e. install gems system-wide).

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **chef_gem** resource in a recipe is as follows:

```
chef_gem "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `chef_gem` tells the chef-client to use the `Chef::Provider::Package::Rubygems` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:reconfig`	Use to reconfigure a package. This action requires a response file.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Rubygems`	`chef_gem`	Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a gems file for use in recipes

```
chef_gem "right_aws" do
  action :install
end

require 'right_aws'
```

Install MySQL for Chef

```
execute "apt-get update" do
  ignore_failure true
  action :nothing
end.run_action(:run) if node['platform_family'] == "debian"

node.set['build_essential']['compiletime'] = true
include_recipe "build-essential"
include_recipe "mysql::client"

node['mysql']['client']['packages'].each do |mysql_pack|
  resources("package[#{mysql_pack}]").run_action(:install)
end

chef_gem "mysql"
```

chef_handler

Use the **chef_handler** resource to enable handlers during a chef-client run. The resource allows arguments to be passed to the chef-client, which then applies the conditions defined by the custom handler to the node attribute data collected during the chef-client run, and then processes the handler based on that data.

The **chef_handler** resource is typically defined early in a node's run-list (often being the first item). This ensures that all of the handlers will be available for the entire chef-client run.

The **chef_handler** resource is included with the **chef_handler** cookbook. This cookbook defines the the resource itself and also provides the location in which the chef-client looks for custom handlers. All custom handlers should be added to the `files/default/handlers` directory in the **chef_handler** cookbook.

Handler Types

There are three types of handlers:

Handler	Description
exception	An exception handler is used to identify situations that have caused a chef-client run to fail. An exception handler can be loaded at the start of a chef-client run by adding a recipe that contains the `chef_handler` resource to a node's run-list. An exception handler runs when the `failed?` property for the `run_status` object returns `true`.
report	A report handler is used when a chef-client run succeeds and reports back on certain details about that chef-client run. A report handler can be loaded at the start of a chef-client run by adding a recipe that contains the `chef_handler` resource to a node's run-list. A report handler runs when the `success?` property for the `run_status` object returns `true`.
start	A start handler is used to run events at the beginning of the chef-client run. A start handler can be loaded at the start of a chef-client run by adding the start handler to the `start_handlers` setting in the client.rb file or by installing the gem that contains the start handler by using the **chef_gem** resource in a recipe in the **chef-client** cookbook. (A start handler may not be loaded using the `chef_handler` resource.)

Exception / Report

Exception and report handlers are used to trigger certain behaviors in response to specific situations, typically identified during a chef-client run.

- An exception handler is used to trigger behaviors when a defined aspect of a chef-client run fails.
- A report handler is used to trigger behaviors when a defined aspect of a chef-client run is successful.

Both types of handlers can be used to gather data about a chef-client run and can provide rich levels of data about all types of usage, which can be used later for trending and analysis across the entire organization.

Exception and report handlers are made available to the chef-client run in one of the following ways:

- By adding the `chef_handler` resource to a recipe, and then adding that recipe to the run-list for a node. (The `chef_handler` resource is available from the **chef_handler** cookbook.)
- By adding the handler to one of the following settings in the node's client.rb file: `exception_handlers` and/or `report_handlers`

The **chef_handler** resource allows exception and report handlers to be enabled from within recipes, which can then added to the run-list for any node on which the exception or report handler should run. The **chef_handler** resource is available from the **chef_handler** cookbook.

To use the **chef_handler** resource in a recipe, add code similar to the following:

```
chef_handler "name_of_handler" do
  source "/path/to/handler/handler_name"
  action :enable
end
```

For example, a handler for Growl needs to be enabled at the beginning of the chef-client run:

```
chef_gem "chef-handler-growl"
```

and then is activated in a recipe by using the **chef_handler** resource:

```
chef_handler "Chef::Handler::Growl" do
  source "chef/handler/growl"
  action :enable
end
```

Start

A start handler is not loaded into the chef-client run from a recipe, but is instead listed in the client.rb file using the `start_handlers` attribute. The start handler must be installed on the node and be available to the chef-client prior to the start of the chef-client run. Use the **chef-client** cookbook to install the start handler.

Start handlers are made available to the chef-client run in one of the following ways:

- By adding a start handler to the **chef-client** cookbook, which installs the handler on the node so that it is available to the chef-client at the start of the chef-client run
- By adding the handler to one of the following settings in the node's client.rb file: `start_handlers`

The **chef-client** cookbook can be configured to automatically install and configure gems that are required by a start handler. For example:

```
node.set['chef_client']['load_gems']['chef-reporting'] = {
  :require_name => 'chef_reporting',
  :action => :install
}

node.set['chef_client']['start_handlers'] = [
  {
    :class => "Chef::Reporting::StartHandler",
    :arguments => []
  }
]

include_recipe "chef-client::config"
```

Syntax

The syntax for using the **chef_handler** resource in a recipe is as follows:

```
chef_handler "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

Actions

This resource has the following actions:

Action	Description
`:enable`	Use to enable the handler for the current chef-client run on the current node.
`:disable`	Use to disable the handler for the current chef-client run on the current node.

Attributes

This resource has the following attributes:

Attribute	Description
`class_name`	The name of the handler class. This can be module name-spaced.
`source`	The full path to the handler file or the path to a gem (if the handler ships as part of a Ruby gem).
`arguments`	An array of arguments that are passed to the initializer for the handler class. Default value: `[]`. For example: `arguments :key1 => 'val1'` or: `arguments [:key1 => 'val1', :key2 => 'val2']`
`supports`	The type of handler. Possible values: `:exception`, `:report`, or `:start`. Default value: `{ :report => true, :exception => true }`.

Custom Handlers

A custom handler can be created to support any situation. The easiest way to build a custom handler:

1. Download the **chef_handler** cookbook
2. Create a custom handler
3. Write a recipe using the **chef_handler** resource
4. Add that recipe to a node's run-list, often as the first recipe in that run-list

Syntax

The syntax for a handler can vary, depending on what the the situations the handler is being asked to track, the type of handler being used, and so on. All custom exception and report handlers are defined using Ruby and must be a subclass of the `Chef::Handler` class.

```
require "chef/log"

module ModuleName
  class HandlerName < Chef::Handler
    def report
      # Ruby code goes here
    end
  end
end
```

where:

- `require` ensures that the logging functionality of the chef-client is available to the handler
- `ModuleName` is the name of the module as it exists within the `Chef` library
- `HandlerName` is the name of the handler as it is used in a recipe
- `report` is an interface that is used to define the custom handler

For example, the following shows a custom handler that sends an email that contains the exception data when a chef-client run fails:

```
require "net/smtp"

module OrgName
  class SendEmail < Chef::Handler
    def report
      if run_status.failed? then
        message  = "From: sender_name <sender@example.com>\n"
        message << "To: recipient_address <recipient@example.com>\n"
        message << "Subject: chef-client Run Failed\n"
        message << "Date: #{Time.now.rfc2822}\n\n"
        message << "Chef run failed on #{node.name}\n"
        message << "#{run_status.formatted_exception}\n"
        message << Array(backtrace).join("\n")
        Net::SMTP.start('your.smtp.server', 25) do |smtp|
          smtp.send_message message, 'sender@example', 'recipient@example'
        end
      end
    end
  end
end
```

and then is used in a recipe like:

```
send_email "blah" do
  # recipe code
end
```

`report` Interface

The `report` interface is used to define how a handler will behave and is a required part of any custom handler. The syntax for the `report` interface is as follows:

```
def report
  # Ruby code
end
```

The Ruby code used to define a custom handler will vary significantly from handler to handler. The chef-client includes two default handlers: `error_report` and `json_file`. Their use of the `report` interface is shown below.

The error_report handler:

```
require 'chef/handler'
require 'chef/resource/directory'

class Chef
  class Handler
    class ErrorReport < ::Chef::Handler
      def report
        Chef::FileCache.store("failed-run-data.json", Chef::JSONCompat.to_json_pretty(data), 0640)
        Chef::Log.fatal("Saving node information to #{Chef::FileCache.load("failed-run-data.json", false)}")
      end
    end
 end
end
```

The json_file handler:

```
require 'chef/handler'
require 'chef/resource/directory'
```

```
class Chef
  class Handler
    class JsonFile < ::Chef::Handler
      attr_reader :config
      def initialize(config={})
        @config = config
        @config[:path] ||= "/var/chef/reports"
        @config
      end
      def report
        if exception
          Chef::Log.error("Creating JSON exception report")
        else
          Chef::Log.info("Creating JSON run report")
        end
        build_report_dir
        savetime = Time.now.strftime("%Y%m%d%H%M%S")
        File.open(File.join(config[:path], "chef-run-report-#{savetime}.json"), "w") do |file|
          run_data = data
          run_data[:start_time] = run_data[:start_time].to_s
          run_data[:end_time] = run_data[:end_time].to_s
          file.puts Chef::JSONCompat.to_json_pretty(run_data)
        end
      end
      def build_report_dir
        unless File.exists?(config[:path])
          FileUtils.mkdir_p(config[:path])
          File.chmod(00700, config[:path])
        end
      end
    end
  end
end
```

Optional Interfaces

The following interfaces may be used in a handler in the same way as the `report` interface to override the default handler behavior in the chef-client. That said, the following interfaces are not typically used in a handler and, for the most part, are completely unnecessary for a handler to work properly and/or as desired.

``data``

The `data` method is used to return the Hash representation of the `run_status` object. For example:

```
def data
  @run_status.to_hash
end
```

``run_report_safely``

The `run_report_safely` method is used to run the report handler, rescuing and logging errors that may arise as the handler runs and ensuring that all handlers get a chance to run during the chef-client run (even if some handlers fail during that run). In general, this method should never be used as an interface in a custom handler unless this default behavior simply must be overridden.

```
def run_report_safely(run_status)
  run_report_unsafe(run_status)
rescue Exception => e
  Chef::Log.error("Report handler #{self.class.name} raised #{e.inspect}")
  Array(e.backtrace).each { |line| Chef::Log.error(line) }
ensure
  @run_status = nil
end
```

``run_report_unsafe``

The `run_report_unsafe` method is used to run the report handler without any error handling. This method should never be used directly in any handler, except during testing of that handler. For example:

```
def run_report_unsafe(run_status)
  @run_status = run_status
  report
end
```

`run_status` Object

The `run_status` object is initialized by the chef-client before the `report` interface is run for any handler. The `run_status` object keeps track of the status of the chef-client run and will contain some (or all) of the following properties:

Property	Description
`all_resources`	A list of all resources that are included in the `resource_collection` property for the current chef-client run.

Property	Description
`backtrace`	A backtrace associated with the uncaught exception data which caused a chef-client run to fail, if present; `nil` for a successful chef-client run.
`elapsed_time`	The amount of time between the start (`start_time`) and end (`end_time`) of a chef-client run.
`end_time`	The time at which a chef-client run ended.
`exception`	The uncaught exception data which caused a chef-client run to fail; `nil` for a successful chef-client run.
`failed?`	Use to show that a chef-client run has failed when uncaught exceptions were raised during a chef-client run. An exception handler runs when the `failed?` indicator is `true`.
`node`	The node on which the chef-client run occurred.
`run_context`	An instance of the `Chef::RunContext` object; used by the chef-client to track the context of the run; provides access to the `cookbook_collection`, `resource_collection`, and `definitions` properties.
`start_time`	The time at which a chef-client run started.
`success?`	Use to show that a chef-client run succeeded when uncaught exceptions were not raised during a chef-client run. A report handler runs when the `success?` indicator is `true`.
`updated_resources`	A list of resources that were marked as updated as a result of the chef-client run.

Note

These properties are not always available. For example, a start handler runs at the beginning of the chef-client run, which means that properties like `end_time` and `elapsed_time` are still unknown and will be unavailable to the `run_status` object.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Enable the CloudkickHandler handler

The following example shows how to enable the `CloudkickHandler` handler, which adds it to the default handler path and passes the `oauth` key/secret to the handler's initializer:

```
chef_handler "CloudkickHandler" do
  source "#{node['chef_handler']['handler_path']}/cloudkick_handler.rb"
  arguments [node['cloudkick']['oauth_key'], node['cloudkick']['oauth_secret']]
  action :enable
end
```

Enable handlers during the compile phase

```
chef_handler "Chef::Handler::JsonFile" do
  source "chef/handler/json_file"
  arguments :path => '/var/chef/reports'
  action :nothing
end.run_action(:enable)
```

Handle only exceptions

```
chef_handler "Chef::Handler::JsonFile" do
  source "chef/handler/json_file"
  arguments :path => '/var/chef/reports'
  supports :exception => true
  action :enable
end
```

Cookbook Versions (a custom handler)

Community member `juliandunn` created a custom report handler that logs all of the cookbooks and cookbook versions that were used during the chef-client run, and then reports after the run is complete. This handler requires the **chef_handler** resource (which is available from the **chef_handler** cookbook).

cookbook_versions.rb:

The following custom handler defines how cookbooks and cookbook versions that are used during the chef-client run will be compiled into a report using the `Chef::Log` class in the chef-client:

```
require 'chef/log'

module Opscode
  class CookbookVersionsHandler < Chef::Handler

    def report
      cookbooks = run_context.cookbook_collection
      Chef::Log.info("Cookbooks and versions run: #{cookbooks.keys.map {|x| cookbooks[x].name.to_s + " " + c
    end
  end
end
```

default.rb:

The following recipe is added to the run-list for every node on which a list of cookbooks and versions will be generated as report output after every chef-client run.

```
include_recipe "chef_handler"

cookbook_file "#{node["chef_handler"]["handler_path"]}/cookbook_versions.rb" do
  source "cookbook_versions.rb"
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end

chef_handler "Opscode::CookbookVersionsHandler" do
  source "#{node["chef_handler"]["handler_path"]}/cookbook_versions.rb"
  supports :report => true
  action :enable
end
```

This recipe will generate report output similar to the following:

```
[2013-11-26T03:11:06+00:00] INFO: Chef Run complete in 0.300029878 seconds
[2013-11-26T03:11:06+00:00] INFO: Running report handlers
[2013-11-26T03:11:06+00:00] INFO: Cookbooks and versions run: ["chef_handler 1.1.4", "cookbook_versions_hand
[2013-11-26T03:11:06+00:00] INFO: Report handlers complete
```

JsonFile Handler

The json_file handler is available from the `chef_handler` cookbook and can be used with exceptions and reports. It serializes run status data to a JSON file. This handler may be enabled in one of the following ways.

By adding the following lines of Ruby code to either the client.rb file or the solo.rb file, depending on how the chef-client is being run:

```
require 'chef/handler/json_file'
report_handlers << Chef::Handler::JsonFile.new(:path => "/var/chef/reports")
exception_handlers << Chef::Handler::JsonFile.new(:path => "/var/chef/reports")
```

By using the chef_handler resource in a recipe, similar to the following:

```
chef_handler "Chef::Handler::JsonFile" do
  source "chef/handler/json_file"
  arguments :path => '/var/chef/reports'
  action :enable
end
```

After it has run, the run status data can be loaded and inspected via Interactive Ruby (IRb):

```
irb(main):001:0> require 'rubygems' => true
irb(main):002:0> require 'json' => true
irb(main):003:0> require 'chef' => true
irb(main):004:0> r = JSON.parse(IO.read("/var/chef/reports/chef-run-report-20110322060731.json")) => ... out
irb(main):005:0> r.keys => ["end_time", "node", "updated_resources", "exception", "all_resources", "success"
irb(main):006:0> r['elapsed_time'] => 0.00246
```

Register the JsonFile handler

```
chef_handler "Chef::Handler::JsonFile" do
  source "chef/handler/json_file"
  arguments :path => '/var/chef/reports'
  action :enable
end
```

ErrorReport Handler

The error_report handler is built into the chef-client and can be used for both exceptions and reports. It serializes error report data to a JSON file. This handler may be enabled in one of the following ways.

By adding the following lines of Ruby code to either the client.rb file or the solo.rb file, depending on how the chef-client is being run:

```
require 'chef/handler/error_report'
report_handlers << Chef::Handler::ErrorReport.new()
exception_handlers << Chef::Handler::ErrorReport.new()
```

By using the chef_handler resource in a recipe, similar to the following:

```
chef_handler "Chef::Handler::ErrorReport" do
  source "chef/handler/error_report"
  action :enable
end
```

cookbook_file

Use the **cookbook_file** resource to transfer files from a sub-directory of `COOKBOOK_NAME/files/` to a specified path located on a host that is running the chef-client. The file is selected according to file specificity, which allows different source files to be used based on the hostname, host platform (operating system, distro, or as appropriate), or platform version. Files that are located in the `COOKBOOK_NAME/files/default` sub-directory may be used on any platform.

During a chef-client run, the checksum for each local file is calculated and then compared against the checksum for the same file as it currently exists in the cookbook on the Chef server. A file is not transferred when the checksums match. Only files that require an update are transferred from the Chef server to a node.

Syntax

The syntax for using the **cookbook_file** resource in a recipe is as follows:

```
cookbook_file "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `cookbook_file` tells the chef-client to use the `Chef::Provider::CookbookFile` provider during the chef-client run
- `name` is the name of the resource block; when the `source` attribute is not specified as part of a recipe, `name` is also the path to the `/files` directory in a cookbook
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

The following is an example of how the **cookbook_file** resource can work when used in a recipe. In this example, because the `source` attribute is unspecified, the name of the resource (`"cookbook_test_file"`) defines the name the source file. The chef-client will look for this source file in the `/cookbook_name/files/default/` directory. The `path` attribute defines the location in which the file will be created. The `:create_if_missing` action ensures that nothing happens if the file already exists.

```
cookbook_file "cookbook_test_file" do
  path "/tmp/test_file"
  action :create_if_missing
end
```

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a file. If a file already exists (but does not match), use to update that file to match.
`:create_if_missing`	Use to create a file only if the file does not exist. (When the file exists, nothing happens.)
`:delete`	Use to delete a file.
`:touch`	Use to touch a file. This updates the access (atime) and file modification (mtime) times for a file. (This action may be used with this resource, but is typically only used with the **file** resource.)

Attributes

This resource has the following attributes:

Attribute	Description
`atomic_update`	Use to perform atomic file updates on a per-resource basis. Set to `true` for atomic file updates. Set to `false` for non-atomic file updates. (This setting overrides `file_atomic_update`, which is a global

Attribute	Description
	setting found in the client.rb file.) Default value: `true`.
`backup`	The number of backups to be kept. Set to `false` to prevent backups from being kept. Default value: `5`.
`cookbook`	The cookbook in which a file is located (if it is not located in the current cookbook). The default value is the current cookbook.
`force_unlink`	Use to specify how the chef-client handles certain situations when the target file turns out not to be a file. For example, when a target file is actually a symlink. Set to `true` to have the chef-client delete the non-file target and replace it with the specified file. Set to `false` for the chef-client to raise an error. Default value: `false`.
`group`	A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`manage_symlink_source`	Use to have the chef-client detect and manage the source file for a symlink. Possible values: `nil`, `true`, or `false`. When this value is set to `nil`, the chef-client will manage a symlink's source file and emit a warning. When this value is set to `true`, the chef-client will manage a symlink's source file and not emit a warning. Default value: `nil`. The default value will be changed to `false` in a future version.
`mode`	A quoted string that defines the octal mode for a file. If `mode` is not specified and if the file already exists, the existing mode on the file is used. If `mode` is not specified, the file does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the file will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`owner`	A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`path`	The path to the location in which a file will be created. Using a fully qualified path is recommended, but is not always required. Microsoft Windows: A path that begins with a forward slash (`/`) will point to the root of the current working directory of the chef-client process. This path can vary from system to system. Therefore, using a path that begins with a forward slash (`/`) is not recommended.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`rights`	Microsoft Windows only. The permissions for users and groups in a Microsoft Windows environment. For example: `rights <permissions>, <principal>, <options>` where `<permissions>` specifies the rights granted to the principal, `<principal>` is the group or user name, and `<options>` is a Hash with one (or more) advanced rights options.
`source`	The location of a file in the `/files` directory in a cookbook located in the chef-repo. Can be used to distribute specific files to specific platforms. (See "File Specificity" below for more information.) Default value: the `name` of the resource block. (See "Syntax" section above for more information.)

Note

Use the `owner` and `right` attributes and avoid the `group` and `mode` attributes whenever possible. The `group` and `mode` attributes are not true Microsoft Windows concepts and are provided more for backward compatibility than for best practice.

Warning

For a machine on which SELinux is enabled, the chef-client will create files that correctly match the default policy settings only when the cookbook that defines the action also conforms to the same policy.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::CookbookFile`	`cookbook_file`	The default provider for all platforms.

File Specificity

A cookbook is frequently designed to work across many platforms and is often required to distribute a specific file to a specific platform. A cookbook can be designed to support the distribution of files across platforms, while ensuring that the correct file ends up on each system.

The pattern for file specificity is as follows:

1. host-node[:fqdn]
2. node[:platform]-node[:platform_version]
3. node[:platform]-version_components: The version string is split on decimals and searched from greatest specificity to least; for example, if the location from the last rule was centos-5.7.1, then centos-5.7 and centos-5 would also be searched.
4. node[:platform]
5. default

A cookbook may have a `/files` directory structure like this:

```
files/
   host-foo.example.com
   ubuntu-10.04
   ubuntu-10
   ubuntu
   redhat-5.8
   redhat-6.4
   ...
   default
```

and a resource that looks something like the following:

```
cookbook_file "/usr/local/bin/apache2_module_conf_generate.pl" do
  source "apache2_module_conf_generate.pl"
  mode '0755'
  owner 'root'
  group 'root'
end
```

This resource is matched in the same order as the `/files` directory structure. For a node that is running Ubuntu 10.04, the second item would be the matching item and the location to which the file identified in the **cookbook_file** resource would be distributed:

```
host-foo.example.com/apache2_module_conf_generate.pl
ubuntu-10.04/apache2_module_conf_generate.pl
ubuntu-10/apache2_module_conf_generate.pl
ubuntu/apache2_module_conf_generate.pl
default/apache2_module_conf_generate.pl
```

If the `apache2_module_conf_generate.pl` file was located in the cookbook directory under `files/host-foo.example.com/`, the specified file(s) would only be copied to the machine with the domain name foo.example.com.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Transfer a file

```
cookbook_file "/tmp/testfile" do
  source "testfile"
  mode '0644'
end
```

Handle cookbook_file and yum_package resources in the same recipe

When a **cookbook_file** resource and a **yum_package** resource are both called from within the same recipe, use the `flush_cache` attribute to dump the in-memory Yum cache, and then use the repository immediately to ensure that the correct package is installed:

```
cookbook_file "/etc/yum.repos.d/custom.repo" do
```

```
  source "custom"
  mode '0644'
end

yum_package "only-in-custom-repo" do
  action :install
  flush_cache [:before]
end
```

Install repositories from a file, trigger a command, and force the internal cache to reload

The following example shows how to install new Yum repositories from a file, where the installation of the repository triggers a creation of the Yum cache that forces the internal cache for the chef-client to reload:

```
execute "create-yum-cache" do
 command "yum -q makecache"
 action :nothing
end

ruby_block "reload-internal-yum-cache" do
  block do
    Chef::Provider::Package::Yum::YumCache.instance.reload
  end
  action :nothing
end

cookbook_file "/etc/yum.repos.d/custom.repo" do
  source "custom"
  mode '0644'
  notifies :run, "execute[create-yum-cache]", :immediately
  notifies :create, "ruby_block[reload-internal-yum-cache]", :immediately
end
```

Use a case statement

The following example shows how a case statement can be used to handle a situation where an application needs to be installed on multiple platforms, but where the install directories are different paths, depending on the platform:

```
cookbook_file "application.pm" do
  path case node['platform']
    when "centos","redhat"
      "/usr/lib/version/1.2.3/dir/application.pm"
    when "arch"
      "/usr/share/version/core_version/dir/application.pm"
    else
      "/etc/version/dir/application.pm"
    end
  source "application-#{node['languages']['perl']['version']}.pm"
  owner 'root'
  group 'root'
  mode '0644'
end
```

cron

Use the **cron** resource to manage cron entries for time-based job scheduling. Attributes for a schedule will default to `*` if not provided. The **cron** resource requires access to a crontab program, typically cron.

Warning

The **cron** resource should only be used to modify an entry in a crontab file. Use the **cookbook_file** or **template** resources to add a crontab file to the cron.d directory. The `cron_d` lightweight resource (found in the cron cookbook) is another option for managing crontab files.

Syntax

The syntax for using the **cron** resource in a recipe is as follows:

```
cron "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `cron` tells the chef-client to use the `Chef::Provider::Cron` provider during the chef-client run
- `"name"` is the name of the cron entry
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example, weekly cookbook reports:

```
cron "cookbooks_report" do
  action node.tags.include?('cookbooks-report') ? :create : :delete
  minute '0'
  hour '0'
  weekday '1'
  user "getchef"
  mailto "nharvey@getchef.com"
  home "/srv/supermarket/shared/system"
  command %Q{
    cd /srv/supermarket/current &&
    env RUBYLIB="/srv/supermarket/current/lib"
    RAILS_ASSET_ID=`git rev-parse HEAD` RAILS_ENV="#{rails_env}"
    bundle exec rake cookbooks_report
  }
end
```

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create an entry in a cron table file (“crontab”). If an entry already exists (but does not match), use to update that entry to match.
`:delete`	Use to delete an entry from a cron table file (“crontab”).

Attributes

This resource has the following attributes:

Attribute	Description

`command`	The command to be run or the path to a file that contains the command to be run.
	Some examples:
	`command if [ -x /usr/share/mdadm/checkarray ] && [ $(date +\%d) -le 7 ];`
	`then /usr/share/mdadm/checkarray --cron --all --idle --quiet; fi`
	and:
	`command %Q{`
	`  cd /srv/opscode-community-site/current &&`
	`  env RUBYLIB="/srv/opscode-community-site/current/lib"`
	``  RAILS_ASSET_ID=`git rev-parse HEAD` RAILS_ENV="#{rails_env}"``
	`  bundle exec rake cookbooks_report`
	`}`
	and:
	`command "/srv/app/scripts/daily_report"`

`day`	The day of month at which the cron entry should run (1 - 31). Default value: `*`.
`home`	Use to set the `HOME` environment variable.
`hour`	The hour at which the cron entry should run (0 - 23). Default value: `*`.
`mailto`	Use to set the `MAILTO` environment variable.
`minute`	The minute at which the cron entry should run (0 - 59). Default value: `*`.
`month`	The month in the year on which a cron entry should run (1 - 12). Default value: `*`.
`path`	Use to set the `PATH` environment variable.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`shell`	Use to set the `SHELL` environment variable.
`user`	The name of the user that runs the command. If the `user` attribute is changed, the original `user` for the crontab program will continue to run until that crontab program is deleted. Default value: `root`.
`weekday`	The day of the week on this entry should run (0 - 6), where Sunday = 0. Default value: `*`. May be entered as a symbol, e.g. `:monday` or `:friday`.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Cron`	`cron`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Run a program at a specified interval

```
cron "noop" do
  hour '5'
  minute '0'
  command "/bin/true"
end
```

Run an entry if a folder exists

```
cron "ganglia_tomcat_thread_max" do
  command "/usr/bin/gmetric -n 'tomcat threads max' -t uint32 -v `/usr/local/bin/tomcat-stat --thread-max`"
  only_if do File.exist?("/home/jboss") end
end
```

Run every Saturday, 8:00 AM

The following example shows a schedule that will run every hour at 8:00 each Saturday morning, and will then send an email to "admin@opscode.com" after each run.

```
cron "name_of_cron_entry" do
  minute '0'
  hour '8'
  weekday '6'
  mailto "admin@opscode.com"
  action :create
end
```

Run only in November

The following example shows a schedule that will run at 8:00 PM, every weekday (Monday through Friday), but only in November:

```
cron "name_of_cron_entry" do
  minute '0'
  hour '20'
  day '*'
  month '11'
  weekday '1-5'
  action :create
end
```

csh

Use the **csh** resource to execute scripts using the csh interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **csh** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **csh** resource in a recipe is as follows:

```
csh "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `csh` tells the chef-client to use the `Chef::Resource::Script::Csh` provider during the chef-client run

- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Csh`	`csh`	The provider that is used with the csh command interpreter.

Examples

None.

deploy

Use the **deploy** resource to manage and control deployments. This is a popular resource, but is also complex, having the most attributes, multiple providers, the added complexity of callbacks, plus four attributes that support layout modifications from within a recipe.

The **deploy** resource is modeled after Capistrano, a utility and framework for executing commands in parallel on multiple remote machines via SSH. The **deploy** resource is designed to behave in a way that is similar to the `deploy` and `deploy:migration` tasks in Capistrano.

Syntax

The syntax for using the **deploy** resource in a recipe is as follows:

```
deploy "name" do
  attribute "value" # see attributes section below
  ...
  callback do
    # callback, including release_path or new_resource
  end
  ...
  purge_before_symlink
  create_dirs_before_symlink
  symlink
  action :action # see actions section below
end
```

where

- `deploy` tells the chef-client to use either the `Chef::Provider::Deploy::Revision` or `Chef::Provider::Deploy::Timestamped` provider during the chef-client run. More specific short names—`timestamped_deploy`, `deploy_revision`, or `deploy_branch`—can be used instead of the `deploy` short name.
- `name` is the name of the resource block; when the `deploy_to` attribute is not specified as part of a recipe, `name` is also the location in which the deployment steps will occur
- `attribute` is zero (or more) of the attributes that are available for this resource
- `callback` represents additional Ruby code that is used to pass a block or to specify a file, and then provide additional information to the chef-client at specific times during the deployment process
- `purge_before_symlink`, `create_dirs_before_symlink`, and `symlink` are attributes that are used to link configuration files, remove directories, create directories, or map files and directories during the deployment process
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example, an application that is deployed to a folder named `"/path/to/application"`:

```
deploy_revision "/path/to/application" do
  repo 'ssh://name-of-git-repo/repos/repo.git'
  migrate false
  purge_before_symlink %w{one two folder/three}
  create_dirs_before_symlink []
  symlinks(
    "one"   => "one",
    "two"   => "two",
    "three" => "folder/three"
  )
  before_restart do
    # some Ruby code
  end
  notifies :restart, "service[foo]"
  notifies :restart, "service[bar]"
end
```

For the example shown above:

- Because an action is not explicitly specified, the chef-client will use the default action: `:deploy`
- The `purge_before_symlink` application layout is an array of paths that will be cleared before the `symlinks` attribute is run
- The `create_dirs_before_symlink` attribute is empty, which is different from the default
- The `symlinks` attribute is creating three symbolic links
- The `before_restart` callback is being used to add custom actions that will occur at the end of the deployment process, but before any services have been notified
- At the end, the recipe is using the `notifies` attribute—a common attribute available to all resources—to alert two services (named "foo" and "bar") that they should restart.

Deploy Strategies

In the `deploy` directory, a sub-directory named `shared` must be created. This sub-directory is where configuration and temporary files will be kept. A typical Ruby on Rails application will have `config`, `log`, `pids`, and `system` directories within the `shared` directory to keep the files stored there independent of the code in the source repository.

In addition to the `shared` sub-directory, the deploy process will create sub-directories named `releases` and `current` (also in the `deploy` directory). The `release` directory holds (up to) five most recently deployed versions of an application. The `current` directory holds the currently-released version.

For example:

```
deploy_directory/
  current/
  releases/
  shared/
    config/
```

```
log/
pids/
system/
```

Deploy Cache File

The chef-client uses a cache file to keep track of the order in which each revision of an application is deployed. By default, the cache file is located at `/var/chef/cache/revision-deploys/APPNAME/`. To force a re-deploy, delete the deployment directory or delete the cache file.

Deploy Phases

A deployment happens in four phases:

1. **Checkout**—the chef-client uses the **scm** resource to get the specified application revision, placing a clone or checkout in the sub-directory of the `deploy` directory named `cached-copy`. A copy of the application is then placed in a sub-directory under `releases`.
2. **Migrate**—If a migration is to be run, the chef-client symlinks the database configuration file into the checkout (`config/database.yml` by default) and runs the migration command. For a Ruby on Rails application, the `migration_command` is usually set to `rake db:migrate`.
3. **Symlink**—Directories for shared and temporary files are removed from the checkout (`log`, `tmp/pids`, and `public/system` by default). After this step, any needed directories (`tmp`, `public`, and `config` by default) are created if they don't already exist. This step is completed by symlinking shared directories into the current `release`, `public/system`, `tmp/pids`, and `log` directories, and then symlinking the `release` directory to `current`.
4. **Restart**—The application is restarted according to the restart command set in the recipe.

Callbacks

In-between each step in a deployment process, callbacks can be run using arbitrary Ruby code, including recipes. All callbacks support embedded recipes given in a block, but each callback assumes a shell command (instead of a deploy hook filename) when given a string.

The following callback types are available:

Callback	Description
`after_restart`	A block of code or a path to a file that contains code that is run after restarting. Default value: `deploy/after_restart.rb`.
`before_migrate`	A block of code (or a path to a file that contains code) that is run before a migration. Default value: `deploy/before_migrate.rb`.
`before_restart`	A block of code (or a path to a file that contains code) that is run before restarting. Default value: `deploy/before_restart.rb`.
`before_symlink`	A block of code (or a path to a file that contains code) that is run before symbolic linking. Default value: `deploy/before_symlink.rb`.

Each of these callback types can be used in one of three ways:

- To pass a block of code, such as Ruby or Python
- To specify a file
- To do neither; the chef-client will look for a callback file named after one of the callback types (`before_migrate.rb`, for example) and if the file exists, to evaluate it as if it were a specified file

Within a callback, there are two ways to get access to information about the deployment:

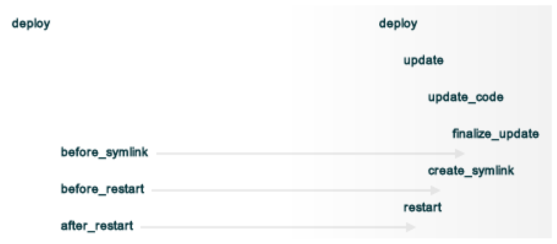
- `release_path` can be used to get the path to the current release
- `new_resource` can be used to access the deploy resource, including environment variables that have been set there (using `new_resource` is a preferred approach over using the `@configuration` variable)

Both of these options must be available at the top-level within the callback, along with any assigned values that will be used later in the callback.

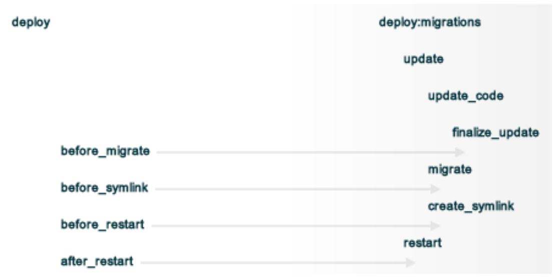
Callbacks and Capistrano

If you are familiar with Capistrano, the following examples should help you know when to use the various callbacks that are available. If you are not familiar with Capistrano, then follow the semantic names of these callbacks to help you determine when to use each of the callbacks within a recipe that is built with the **deploy** resource.

The following example shows where callbacks fit in relation to the steps taken by the `deploy` process in Capistrano:



and the following example shows the same comparison, but with the `deploy:migrations` process:



Layout Modifiers

The **deploy** resource expects an application to be structured like a Ruby on Rails application, but the layout can be modified to meet custom requirements as needed. Use the following attributes within a recipe to modify the layout of a recipe that is using the **deploy** resource:

Layout Modifiers	Description
`create_dirs_before_symlink`	Use this attribute to create directories before symbolic links are created. This attribute runs after `purge_before_symlink` and before `symlink`.
`purge_before_symlink`	Use this attribute to specify an array of directories (relative to the application root) that should be removed from a checkout before symbolic links are created. This attribute runs before `create_dirs_before_symlink` and before `symlink`.
`symlink_before_migrate`	Use this attribute to map files in a shared directory to the current release directory. The symbolic links for these files will be created before any migration is run. Use `symlink_before_migrate({})` or `symlink_before_migrate nil` instead of `symlink_before_migrate {}` because `{}` will be interpreted as a block rather than an empty Hash. Set to `nil` to prevent the creation of default symbolic links.
`symlinks`	Use this attribute to map files in a shared directory to their paths in the current release directory. This attribute runs after `create_dirs_before_symlink` and `purge_before_symlink`.

Actions

This resource has the following actions:

Action	Description
`:deploy`	Default. Use to deploy an application.
`:force_deploy`	Use to remove any existing release of the same code version and re-deploy a new one in its place.
`:rollback`	Use to roll an application back to the previous release.

Attributes

This resource has the following attributes:

Attribute	Description
`after_restart`	A block of code or a path to a file that contains code that is run after restarting. Default value: `deploy/after_restart.rb`.
`before_migrate`	A block of code (or a path to a file that contains code) that is run before a migration. Default value: `deploy/before_migrate.rb`.

Attribute	Description
`before_restart`	A block of code (or a path to a file that contains code) that is run before restarting. Default value: `deploy/before_restart.rb`.
`before_symlink`	A block of code (or a path to a file that contains code) that is run before symbolic linking. Default value: `deploy/before_symlink.rb`.
`branch`	The alias for the revision.
`create_dirs_before_symlink`	Use this attribute to create directories before symbolic links are created. This attribute runs after `purge_before_symlink` and before `symlink`. Default value: `%w{tmp public config}` (or the same as `["tmp", "public", "config"]`).
`deploy_to`	The "meta root" for the application, if different from the path that is used to specify the name of a resource. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`group`	The system group that is responsible for the checked-out code.
`keep_releases`	The number of releases for which a backup is kept. Default value: `5`.
`migrate`	Use to run a migration command. Default value: `false`.
`migration_command`	A string that contains a shell command that can be executed to run a migration operation.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`purge_before_symlink`	Use this attribute to specify an array of directories (relative to the application root) that should be removed from a checkout before symbolic links are created. This attribute runs before `create_dirs_before_symlink` and before `symlink`. Default value: `%w{log tmp/pids public/system}` (or the same as `["log", "tmp/pids", "public/system"]`.
`repo`	The alias for the repository.
`repository`	The URI for the repository.
`repository_cache`	The name of the sub-directory in which the pristine copy of an application's source is kept. Default value: `cached-copy`.
`restart_command`	A string that contains a shell command that can be executed to run a restart operation.
`revision`	The revision to be checked out. This can be symbolic, like `HEAD` or it can be a source control management-specific revision identifier. Default value: `HEAD`.
`rollback_on_error`	Use to roll a resource back to a previously-deployed release if an error occurs when deploying a new release. Default value: `false`.
`scm_provider`	The name of the source control management provider. Default value: `Chef::Provider::Git`. Optional values: `Chef::Provider::Subversion`.
`symlinks`	Use this attribute to map files in a shared directory to their paths in the current release directory. This attribute runs after `create_dirs_before_symlink` and `purge_before_symlink`. Default value: `{"system" => "public/system", "pids" => "tmp/pids", "log" => "log"}`.
`symlink_before_migrate`	Use this attribute to map files in a shared directory to the current release directory. The symbolic links for these files will be created before any migration is run. Use `symlink_before_migrate({})` or `symlink_before_migrate nil` instead of `symlink_before_migrate {}` because `{}` will be interpreted as a block rather than an empty Hash. Set to `nil` to prevent the creation of default symbolic links. Default value: `{"config/database.yml" => "config/database.yml"}`.
`timeout`	The amount of time (in seconds) to wait for a command to execute before timing out. When specified, this value is passed from the **deploy** resource to the **git** or **subversion** resources.
`user`	The system user that is responsible for the checked-out code.

The following attributes are for use with git only:

Attribute	Description
`enable_submodules`	Use to perform a sub-module initialization and update. Default value: `false`.

Attribute	Description
`git_ssh_wrapper`	The alias for the `ssh_wrapper`.
`remote`	The remote repository to be used when synchronizing an existing clone. Default value: `origin`.
`shallow_clone`	Use to set the clone depth to 5. Default value: `false`.
`ssh_wrapper`	The path to the wrapper script used when running SSH with git. The `GIT_SSH` environment variable is set to this.

The following attributes are for use with Subversion only:

Attribute	Description
`svn_arguments`	The extra arguments that are passed to the Subversion command.
`svn_password`	The password for the user that has access to the Subversion repository.
`svn_username`	The user name for a user that has access to the Subversion repository.

For example:

```
deploy "/my/deploy/dir" do
  repo "git@github.com/whoami/project"
  revision "abc123" # or "HEAD" or "TAG_for_1.0" or (subversion) "1234"
  user "deploy_ninja"
  enable_submodules true
  migrate true
  migration_command "rake db:migrate"
  environment "RAILS_ENV" => "production", "OTHER_ENV" => "foo"
  shallow_clone true
  keep_releases 10
  action :deploy # or :rollback
  restart_command "touch tmp/restart.txt"
  git_ssh_wrapper "wrap-ssh4git.sh"
  scm_provider Chef::Provider::Git # is the default, for svn: Chef::Provider::Subversion
end
```

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Deploy`	`deploy`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Deploy::Branch`	`deploy_branch`	See below for more information.
`Chef::Provider::Deploy::Revision`	`deploy_revision`	See below for more information.
`Chef::Provider::Deploy::TimestampedDeploy`	`timestamped_deploy`	The default provider for all platforms. See below for more information.

deploy_branch

The `deploy_branch` resource is used in the same way as the `deploy_resource` resource. It uses the `Deploy::Revision` provider and has uses the same set of actions and attributes.

deploy_revision

The `deploy_revision` provider is the recommended provider, even if it is not listed as the default. The `deploy_revision` provider is used to ensure that the name of a release sub-directory is based on a revision identifier. For users of git, this will be the familiar SHA checksum. For users of Subversion, it will be the integer revision number. If a name other than a revision identifier is provided—branch names, tags, and so on—the chef-client will ignore the alternate names and will look up the revision identifier and use it to name the release sub-directory. When the `deploy_revision` provider is given an exact revision to deploy, it will behave in an idempotent manner.

The `deploy_revision` provider results in deployed components under the destination location that is owned by the user who runs the application. This is sometimes an issue for certain workflows. If issues arise, consider the following:

- Incorporate changing permissions to the desired end state from within a recipe
- Add a `before_restart` block to fix up the permissions
- Have an unprivileged user (for example: `opscode`) be the owner of the `deploy` directory and another unprivileged user (for example: `opscodeapp`) run the application. Most often, this is the solution that works best

When using the `deploy_revision` provider, and when the deploy fails for any reason, and when the same code is used to re-deploy, the action should be set manually to `:force_deploy`. Forcing the re-deploy will remove the old release directory, after which the deploy can proceed as usual. (Forcing a re-deploy over the current release can cause some downtime.) Deployed revisions are stored in `(file_cache_path)/revision-deploys/(deploy_path)`.

timestamped_deploy

The `timestamped_deploy` provider is the default **deploy** provider. It is used to name release directories with a timestamp in the form of `YYYYMMDDHHMMSS`. For example: `/my/deploy/dir/releases/20121120162342`. The **deploy** resource will determine whether or not to deploy code based on the existence of the release directory in which it is attempting to deploy. Because the timestamp is different for every chef-client run, the `timestamped_deploy` provider is not idempotent. When the `timestamped_deploy` provider is used, it requires that the action setting on a resource be managed manually in order to prevent unintended continuous deployment.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Modify the layout of a Ruby on Rails application

The layout of the **deploy** resource matches a Ruby on Rails app by default, but this can be customized. To customize the layout, do something like the following:

```
deploy "/my/apps/dir/deploy" do
  # Use a local repo if you prefer
  repo "/path/to/gitrepo/typo/"
  environment "RAILS_ENV" => "production"
  revision "HEAD"
  action :deploy
  migration_command "rake db:migrate --trace"
  migrate true
  restart_command "touch tmp/restart.txt"
  create_dirs_before_symlink  %w{tmp public config deploy}

  # You can use this to customize if your app has extra configuration files
  # such as amqp.yml or app_config.yml
  symlink_before_migrate  "config/database.yml" => "config/database.yml"

  # If your app has extra files in the shared folder, specify them here
  symlinks  "system" => "public/system",
            "pids" => "tmp/pids",
            "log" => "log",
            "deploy/before_migrate.rb" => "deploy/before_migrate.rb",
            "deploy/before_symlink.rb" => "deploy/before_symlink.rb",
            "deploy/before_restart.rb" => "deploy/before_restart.rb",
            "deploy/after_restart.rb" => "deploy/after_restart.rb"
end
```

Use resources within callbacks

Using resources from within your callbacks as blocks or within callback files distributed with your application's source code. To use embedded recipes for callbacks:

```
deploy "#{node['tmpdir']}/deploy" do
  repo "#{node['tmpdir']}/gitrepo/typo/"
  environment "RAILS_ENV" => "production"
  revision "HEAD"
  action :deploy
  migration_command "rake db:migrate --trace"
  migrate true

  # Callback awesomeness:
  before_migrate do
    current_release = release_path

    directory "#{current_release}/deploy" do
      mode '0755'
    end

    # creates a callback for before_symlink
    template "#{current_release}/deploy/before_symlink_callback.rb" do
      source "embedded_recipe_before_symlink.rb.erb"
      mode '0644'
    end

  end

  # This file can contain Chef recipe code, plain ruby also works
  before_symlink "deploy/before_symlink_callback.rb"

  restart do
    current_release = release_path
    file "#{release_path}/tmp/restart.txt" do
```

```
      mode '0644'
    end
  end

end
```

Deploy from a private git repository without using the application cookbook

To deploy from a private git repository without using the **application** cookbook, first ensure that:

- the private key does not have a passphrase, as this will pause a chef-client run to wait for input
- an SSH wrapper is being used
- a private key has been added to the node

and then use code like the following to remove a passphrase from a private key:

```
ssh-keygen -p -P 'YOURPASSPHRASE' -N '' -f id_deploy
```

Use an SSH wrapper

To write a recipe that uses an SSH wrapper:

1. Create a file in the `cookbooks/COOKBOOK_NAME/files/default` directory that is named `wrap-ssh4git.sh` and which contains the following:

   ```
   #!/usr/bin/env bash
   /usr/bin/env ssh -o "StrictHostKeyChecking=no" -i "/tmp/private_code/.ssh/id_deploy" $1 $2
   ```

2. Set up the cookbook file.
3. Add a recipe to the cookbook file similar to the following:

   ```
   directory "/tmp/private_code/.ssh" do
     owner 'ubuntu'
     recursive true
   end

   cookbook_file "/tmp/private_code/wrap-ssh4git.sh" do
     source "wrap-ssh4git.sh"
     owner 'ubuntu'
     mode '0700'
   end

   deploy "private_repo" do
     repo "git@github.com:acctname/private-repo.git"
     user "ubuntu"
     deploy_to "/tmp/private_code"
     action :deploy
     ssh_wrapper "/tmp/private_code/wrap-ssh4git.sh"
   end
   ```

   This will deploy the git repository at `git@github.com:acctname/private-repo.git` in the `/tmp/private_code` directory.

Use a callback to include a file that will be passed as a code block

The code in a file that is included in a recipe using a callback is evaluated exactly as if the code had been put in the recipe as a block. Files are searched relative to the current release.

To specify a file that contains code to be used as a block:

```
deploy "/deploy/dir/" do
  # ...

  before_migrate "callbacks/do_this_before_migrate.rb"
end
```

Use a callback to pass a code block

To pass a block of Python code before a migration is run:

```
deploy_revision "/deploy/dir/" do
  # other attributes
  # ...

  before_migrate do
    # release_path is the path to the timestamp dir
    # for the current release
    current_release = release_path

    # Create a local variable for the node so we'll have access to
    # the attributes
    deploy_node = node

    # A local variable with the deploy resource.
```

```
      deploy_resource = new_resource

      python do
        cwd current_release
        user "myappuser"
        code<<-PYCODE
          # Woah, callbacks in python!
          # ...
          # current_release, deploy_node, and deploy_resource are all available
          # within the deploy hook now.
        PYCODE
      end
    end
  end
```

Use the same API for all recipes using the same gem

Any recipes using the `git-deploy` gem can continue using the same API. To include this behavior in a recipe, do something like the following:

```
deploy "/srv/#{appname}" do
  repo "git://github.com/radiant/radiant.git"
  revision "HEAD"
  user "railsdev"
  enable_submodules false
  migrate true
  migration_command "rake db:migrate"
  # Giving a string for environment sets RAILS_ENV, MERB_ENV, RACK_ENV
  environment "production"
  shallow_clone true
  action :deploy
  restart_command "touch tmp/restart.txt"
end
```

Deploy without creating symbolic links to a shared folder

To deploy without creating symbolic links to a shared folder:

```
deploy "/my/apps/dir/deploy" do
  symlinks {}
end
```

When deploying code that is not Ruby on Rails and symbolic links to a shared folder are not wanted, use parentheses `()` or `Hash.new` to avoid ambiguity. For example, using parentheses:

```
deploy "/my/apps/dir/deploy" do
  symlinks({})
end
```

or using `Hash.new`:

```
deploy "/my/apps/dir/deploy" do
  symlinks Hash.new
end
```

Clear a layout modifier attribute

Using the default attribute values for the various resources is the recommended starting point when working with recipes. Then, depending on what each node requires, these default values can be overridden with node-, role-, environment-, and cookbook-specific values. The **deploy** resource has four layout modifiers: `create_dirs_before_symlink`, `purge_before_symlink`, `symlink_before_migrate`, and `symlinks`. Each of these is a Hash that behaves as an attribute of the **deploy** resource. When these layout modifiers are used in a recipe, they appear similar to the following:

```
deploy "name" do
  ...
  symlink_before_migrate       {"config/database.yml" => "config/database.yml"}
  create_dirs_before_symlink   %w{tmp public config}
  purge_before_symlink         %w{log tmp/pids public/system}
  symlinks                     { "system" => "public/system",
                                 "pids" => "tmp/pids",
                                 "log" => "log"
                               }
  ...
end
```

and then what these layout modifiers look like if they were empty:

```
deploy "name" do
  ...
  symlink_before_migrate       nil
  create_dirs_before_symlink   []
  purge_before_symlink         []
  symlinks                     nil
  ...
```

```
end
```

In most cases, using the empty values for the layout modifiers will prevent the chef-client from passing symbolic linking information to a node during the chef-client run. However, in some cases, it may be preferable to ensure that one (or more) of these layout modifiers do not pass any symbolic linking information to a node during the chef-client run at all. Because each of these layout modifiers are a Hash, the `clear` instance method can be used to clear out these values.

To clear the default values for a layout modifier:

```
deploy "name" do
  ...
  symlink_before_migrate.clear
  create_dirs_before_symlink.clear
  purge_before_symlink.clear
  symlinks.clear
  ...
end
```

In general, use this approach carefully and only after it is determined that nil or empty values won't provide the expected result.

directory

Use the **directory** resource to manage a directory, which is a hierarchy of folders that comprises all of the information stored on a computer. The root directory is the top-level, under which the rest of the directory is organized. The **directory** resource uses the `name` attribute to specify the path to a location in a directory. Typically, permission to access that location in the directory is required.

Syntax

The syntax for using the **directory** resource in a recipe is as follows:

```
directory "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `directory` tells the chef-client to use the `Chef::Provider::Directory` provider during the chef-client run
- `name` is the name of the resource block; when the `path` attribute is not specified as part of a recipe, `name` is also the path to the directory, from the root
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
directory "/var/lib/foo" do
  owner 'root'
  group 'root'
  mode '0644'
  action :create
end
```

A variable may be used to define a directory, and then again within the recipe itself:

```
node.default['apache']['dir'] = '/etc/apache2'

directory node['apache']['dir'] do
  owner 'apache'
  group 'apache'
  action :create
end
```

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a directory. If a directory already exists (but does not match), use to update that directory to match.
`:delete`	Use to delete a directory.

Attributes

This resource has the following attributes:

Attribute	Description
`group`	A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`mode`	A quoted string that defines the octal mode for a directory. If `mode` is not specified and if the directory already exists, the existing mode on the directory is used. If `mode` is not specified, the directory does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the directory will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`owner`	A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`path`	The path to the directory. Using a fully qualified path is recommended, but is not always required. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`recursive`	Use to create or delete parent directories recursively. For the `owner`, `group`, and `mode` attributes, the value of this attribute applies only to the leaf directory. Default value: `false`.
`rights`	Microsoft Windows only. The permissions for users and groups in a Microsoft Windows environment. For example: `rights <permissions>, <principal>, <options>` where `<permissions>` specifies the rights granted to the principal, `<principal>` is the group or user name, and `<options>` is a Hash with one (or more) advanced rights options.

Recursive Directories

The **directory** resource can be used to create directory structures, as long as each directory within that structure is created explicitly. This is because the `recursive` attribute only applies `group`, `mode`, and `owner` attribute values to the leaf directory.

A directory structure:

```
/foo
  /bar
    /baz
```

The following example shows a way create a file in the `/baz` directory:

```
directory "/foo/bar/baz" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

But with this example, the `group`, `mode`, and `owner` attribute values will only be applied to `/baz`. Which is fine, if that’s what you want. But most of the time, when the entire `/foo/bar/baz` directory structure is not there, you must be explicit about each directory. For example:

```
%w[ /foo /foo/bar /foo/bar/baz ].each do |path|
  directory path do
    owner 'root'
    group 'root'
    mode '0755'
  end
```

```
end
```

This approach will create the correct hierarchy—`/foo`, then `/bar` in `/foo`, and then `/baz` in `/bar`—and also with the correct attribute values for `group`, `mode`, and `owner`.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Directory`	`directory`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create a directory

```
directory "/tmp/something" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

Create a directory in Microsoft Windows

```
directory "C:\\tmp\\something.txt" do
  rights :full_control, "DOMAIN\\User"
  inherits false
  action :create
end
```

or:

```
directory 'C:\tmp\something.txt' do
  rights :full_control, 'DOMAIN\User'
  inherits false
  action :create
end
```

Note

The difference between the two previous examples is the single- versus double-quoted strings, where if the double quotes are used, the backslash character (`\`) must be escaped using the Ruby escape character (which is a backslash).

Create a directory recursively

```
%w{dir1 dir2 dir3}.each do |dir|
  directory "/tmp/mydirs/#{dir}" do
    mode '0775'
    owner 'root'
    group 'root'
    action :create
    recursive true
  end
end
```

Delete a directory

```
directory "/tmp/something" do
  recursive true
  action :delete
end
```

Set directory permissions using a variable

The following example shows how read/write/execute permissions can be set using a variable named `user_home` and then for owners and groups on any matching node:

```
user_home = "/#{node[:matching_node][:user]}"

directory user_home do
  owner 'node[:matching_node][:user]'
  group 'node[:matching_node][:group]'
  mode '0755'
  action :create
end
```

where `matching_node` represents a type of node. For example, if the `user_home` variable specified `{node[:nginx]...}`, a recipe might look something like this:

```
user_home = "/#{node[:nginx][:user]}"

directory user_home do
  owner 'node[:nginx][:user]'
  group 'node[:nginx][:group]'
  mode '0755'
  action :create
end
```

Set directory permissions for a specific type of node

The following example shows how permissions can be set for the `/certificates` directory on any node that is running Nginx. In this example, permissions are being set for the owner and group as `root`, and then read/write permissions are granted to the root.

```
directory "#{node[:nginx][:dir]}/shared/certificates" do
  owner 'root'
  group 'root'
  mode '0700'
  recursive true
end
```

Reload the configuration

The following example shows how to reload the configuration of a chef-client using the **remote_file** resource to:

- using an if statement to check whether the plugins on a node are the latest versions
- identify the location from which Ohai plugins are stored
- using the `notifies` attribute and a **ruby_block** resource to trigger an update (if required) and to then reload the client.rb file.

```
directory node[:ohai][:plugin_path] do
  owner 'chef'
  recursive true
end

ruby_block "reload_config" do
  block do
    Chef::Config.from_file("/etc/chef/client.rb")
  end
  action :nothing
end

if node[:ohai].key?(:plugins)
  node[:ohai][:plugins].each do |plugin|
    remote_file node[:ohai][:plugin_path] +"/#{plugin}" do
      source plugin
      owner 'chef'
              notifies :run, "ruby_block[reload_config]", :immediately
    end
  end
end
```

dpkg_package

Use the **dpkg_package** resource to manage packages for the dpkg platform. When a package is installed from a local file, it must be added to the node using the **remote_file** or **cookbook_file** resources.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **dpkg_package** resource in a recipe is as follows:

```
dpkg_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `dpkg_package` tells the chef-client to use the `Chef::Provider::Package::Dpkg` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of

the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Dpkg`	`dpkg_package`	The provider that is used with the dpkg platform. Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
dpkg_package "name of package" do
  action :install
end
```

dsc_script

A resource defines the desired state for a single configuration item present on a node that is under management by Chef. A resource collection—one (or more) individual resources—defines the desired state for the entire node. During every chef-client run, the current state of each resource is tested, after which the chef-client will take any steps that are necessary to repair the node and bring it back into the desired state.

Windows PowerShell is a task-based command-line shell and scripting language developed by Microsoft. Windows PowerShell uses a document-oriented approach for managing Microsoft Windows-based machines, similar to the approach that is used for managing UNIX- and Linux-based machines. Windows PowerShell is a tool-agnostic platform that supports using Chef for configuration management.

Desired State Configuration (DSC) is a feature of Windows PowerShell that provides a set of language extensions, cmdlets, and resources that can be used to declaratively configure software. DSC is similar to Chef, in that both tools are idempotent, take similar approaches to the concept of resources, describe the configuration of a system, and then take the steps required to do that configuration. The most important difference between Chef and DSC is that Chef uses Ruby and DSC is exposed as configuration data from within Windows PowerShell.

Many DSC resources are comparable to built-in Chef resources. For example, both DSC and Chef have **file**, **package**, and **service** resources. The **dsc_script** resource is most useful for those DSC resources that do not have a direct comparison to a resource in Chef, such as the `Archive` resource, a custom DSC resource, an existing DSC script that performs an important task, and so on. Use the **dsc_script** resource to embed the code that defines a DSC configuration directly within a Chef recipe.

Note

Windows PowerShell 4.0 is required for using the **dsc_script** resource with Chef.

Note

The WinRM service must be enabled. (Use `winrm quickconfig` to enable the service.)

Syntax

The syntax for using the **dsc_script** resource in a recipe is as follows:

```
dsc_script "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `dsc_script` tells the chef-client that a DSC resource is based on a Windows PowerShell script
- `name` is the name of the configuration within a DSC script; when the `configuration_name` attribute is not specified as part of a recipe, `name` must also be the name of a valid Windows PowerShell cmdlet
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:run`	Default. Use to run the DSC configuration defined as defined in this resource.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	The code for the DSC configuration script. This attribute may not be used in the same recipe as the `command` attribute.
`command`	The path to a valid Windows PowerShell data file that contains the DSC configuration script. This data file must be capable of running independently of Chef and must generate a valid DSC configuration. This attribute may not be used in the same recipe as the `code` attribute.
`configuration_data`	Use to specify the configuration data for the DSC script. The configuration data must be a valid Windows Powershell data file. This attribute may not be used in the same recipe as the `configuration_data_script` attribute.
`configuration_data_script`	The path to a valid Windows PowerShell data file that also contains a node called `localhost`. This attribute may not be used in the same recipe as the `configuration_data` attribute.
`configuration_name`	The name of a valid Windows PowerShell cmdlet. The name may only contain letter (a-z, A-Z), number (0-9), and underscore (_) characters and should start with a letter. The name may not be null or empty. This attribute may not be used in the same recipe as the `code` attribute.
`flags`	Use to pass parameters to the DSC script that is specified by the `command` attribute. Parameters are defined as key-value pairs, where the value of each key is the parameter to pass. This attribute may not be used in the same recipe as the `code` attribute. For example: `flags ({ :EditorChoice => 'emacs', :EditorFlags => '--maximized' })`. Default value: `nil`.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)

Examples

Specify DSC code directly

DSC data can be specified directly in a recipe:

```
dsc_script 'emacs' do
  code <<-EOH
  Environment 'texteditor'
  {
    Name = 'EDITOR'
    Value = 'c:\\emacs\\bin\\emacs.exe'
  }
  EOH
end
```

Specify DSC code using a Windows Powershell data file

Use the `command` attribute to specify the path to a Windows PowerShell data file. For example, the following Windows PowerShell script defines the `DefaultEditor`:

```
Configuration 'DefaultEditor'
{
  Environment 'texteditor'
    {
      Name = 'EDITOR'
      Value = 'c:\emacs\bin\emacs.exe'
    }
}
```

Use the following recipe to specify the location of that data file:

```
dsc_script 'DefaultEditor' do
  command 'c:\dsc_scripts\emacs.ps1'
end
```

Pass parameters to DSC configurations

If a DSC script contains configuration data that takes parameters, those parameters may be passed using the `flags` attribute. For example, the following Windows PowerShell script takes parameters for the `EditorChoice` and `EditorFlags` settings:

```
$choices = @{'emacs' = 'c:\emacs\bin\emacs';'vi' = 'c:\vim\vim.exe';'powershell' = 'powershell_ise.exe'}
  Configuration 'DefaultEditor'
    {
      [CmdletBinding()]
      param
        (
          $EditorChoice,
          $EditorFlags = ''
        )
      Environment 'TextEditor'
      {
        Name = 'EDITOR'
        Value =  "$($choices[$EditorChoice]) $EditorFlags"
      }
    }
```

Use the following recipe to set those parameters:

```
dsc_script 'DefaultEditor' do
  flags ({ :EditorChoice => 'emacs', :EditorFlags => '--maximized' })
  command 'c:\dsc_scripts\editors.ps1'
end
```

Use custom configuration data

Configuration data in DSC scripts may be customized from a recipe. For example, scripts are typically customized to set the behavior for Windows PowerShell credential data types. Configuration data may be specified in one of three ways: by using the `configuration_data` or `configuration_data_script` attributes or by specifying the path to a valid Windows PowerShell data file.

The following example shows how to specify custom configuration data using the `configuration_data` attribute:

```
dsc_script 'BackupUser' do
  configuration_data <<-EOH
    @{
     AllNodes = @(
          @{
          NodeName = "localhost";
          PSDscAllowPlainTextPassword = $true
          })
     }
  EOH
  code <<-EOH
    $user = 'backup'
    $password = ConvertTo-SecureString -String "YourPass$(random)" -AsPlainText -Force
    $cred = New-Object -TypeName System.Management.Automation.PSCredential -ArgumentList $user, $password
    User $user
      {
        UserName = $user
        Password = $cred
        Description = 'Backup operator'
        Ensure = "Present"
        Disabled = $false
        PasswordNeverExpires = $true
        PasswordChangeRequired = $false
      }
    EOH

  configuration_data <<-EOH
    @{
      AllNodes = @(
          @{
          NodeName = "localhost";
          PSDscAllowPlainTextPassword = $true
          })
      }
    EOH
end
```

The following example shows how to specify custom configuration data using the `configuration_name` attribute. For example, the following Windows PowerShell script defines the `vi` configuration:

```
Configuration 'emacs'
  {
    Environment 'TextEditor'
    {
      Name = 'EDITOR'
      Value = 'c:\emacs\bin\emacs.exe'
    }
}

Configuration 'vi'
{
    Environment 'TextEditor'
    {
      Name = 'EDITOR'
      Value = 'c:\vim\bin\vim.exe'
    }
}
```

Use the following recipe to specify that configuration:

```
dsc_script 'EDITOR' do
  configuration_name 'vi'
  command 'c:\dsc_scripts\editors.ps1'
end
```

Using DSC with other Chef resources

The **dsc_script** resource can be used with other resources. The following example shows how to download a file using the **remote_file** resource, and then uncompress it using the DSC `Archive` resource:

```
remote_file "#{Chef::Config[:file_cache_path]}\\DSCResourceKit620082014.zip" do
  source 'http://gallery.technet.microsoft.com/DSC-Resource-Kit-All-c449312d/file/124481/1/DSC%20Resource%20
end

dsc_script 'get-dsc-resource-kit' do
  code <<-EOH
    Archive reskit
    {
      ensure = 'Present'
      path = "#{Chef::Config[:file_cache_path]}\\DSCResourceKit620082014.zip"
      destination = "#{ENV['PROGRAMW6432']}\\WindowsPowerShell\\Modules"
    }
  EOH
end
```

easy_install_package

Use the **easy_install_package** resource to manage packages for the Python platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **easy_install_package** resource in a recipe is as follows:

```
easy_install_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `easy_install_package` tells the chef-client to use the `Chef::Provider::Package::EasyInstall` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`easy_install_binary`	The location of the Easy Install binary.
`module_name`	The name of the module.
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`python_binary`	The location of the Python binary.
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::EasyInstall`	`easy_install_package`	The provider that is used with Python platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
easy_install_package "name of package" do
  action :install
end
```

env

Use the **env** resource to manage environment keys in Microsoft Windows. After an environment key is set, Microsoft Windows must be restarted before the environment key will be available to the Task Scheduler.

Note

On UNIX-based systems, the best way to manipulate environment keys is with the `ENV` variable in Ruby; however, this approach does not have the same permanent effect as using the **env** resource.

Syntax

The syntax for using the **env** resource in a recipe is as follows:

```
env "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `env` tells the chef-client to use the `Chef::Provider::Env::Windows` provider during the chef-client run
- `name` is the name of the resource block; when the `key_name` attribute is not specified as part of a recipe, `name` is also the name of the environment key that is created, deleted, or modified
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create an environment variable. If an environment variable already exists (but does not match), use to update that environment variable to match.
`:delete`	Use to delete an environment variable.
`:modify`	Use to modify an existing environment variable. This will append the new value to the existing value, using the delimiter specified by the `delim` attribute.

Attributes

This resource has the following attributes:

Attribute	Description
`delim`	The delimiter that is used to separate multiple values for a single key.
`key_name`	The name of the key that will be created, deleted, or modified. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`value`	The value with which `key_name` is set.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Env::Windows`	`env`	The default provider for all Microsoft Windows platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Set an environment variable

```
env "ComSpec" do
  value "C:\\Windows\\system32\\cmd.exe"
end
```

erl_call

Use the **erl_call** resource to connect to a node located within a distributed Erlang system. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The `erl_call` command needs to be on the path for this resource to work properly.

Syntax

The syntax for using the **erl_call** resource in a recipe is as follows:

```
erl_call "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `erl_call` tells the chef-client to use the `Chef::Provider::ErlCall` provider during the chef-client run
- `"name"` is the name of the call
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:run`	Default. Use to run the Erlang call.
`:nothing`	Use to prevent the Erlang call from running.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	The code to be executed on a node located within a distributed Erlang system. Default value: `q()`.
`cookie`	The magic cookie for the node to which a connection is made.
`distributed`	Use to specify that a node is a distributed Erlang node. Default value: `false`.
`name_type`	Use to specify the `node_name` attribute as a short node name (`sname`) or a long node name (`name`). A node with a long node name cannot communicate with a node with a short node name. Default value: `sname`.
`node_name`	The hostname to which the node will connect. Default value: `chef@localhost`.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::ErlCall`	`erl_call`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Run a command

```
erl_call "list names" do
  code "net_adm:names()."
  distributed true
  node_name "chef@latte"
end
```

execute

Use the **execute** resource to execute a command. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

Use the **script** resource to execute a script using a specific interpreter (Ruby, Python, Perl, csh, or Bash).

Syntax

The syntax for using the **execute** resource in a recipe is as follows:

```
execute "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `execute` tells the chef-client to use the `Chef::Provider::Execute` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example, use a whitespace array to identify the names of the pets to be fed:

```
%w{rover fido bubbers}.each do |pet_name|
  execute "feed_pet_#{pet_name}" do
    command "echo 'Feeding: #{pet_name}'; touch '/tmp/#{pet_name}'"
    not_if { ::File.exists?("/tmp/#{pet_name}")}
  end
end
```

Actions

This resource has the following actions:

Action	Description
`:run`	Default. Use to run a command.
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.

Attributes

This resource has the following attributes:

Attribute	Description
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory from which a command is run.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command’s environment $PATH. The default value uses the system path.

Attribute	Description
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Execute`	`execute`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Run a command upon notification

```
execute "slapadd" do
  command "slapadd < /tmp/something.ldif"
  creates "/var/lib/slapd/uid.bdb"
  action :nothing
end

template "/tmp/something.ldif" do
  source "something.ldif"
  notifies :run, "execute[slapadd]", :immediately
end
```

Run a touch file only once while running a command

```
execute "upgrade script" do
  command "php upgrade-application.php && touch /var/application/.upgraded"
  creates "/var/application/.upgraded"
  action :run
end
```

Run a command which requires an environment variable

```
execute "slapadd" do
  command "slapadd < /tmp/something.ldif"
  creates "/var/lib/slapd/uid.bdb"
  action :run
  environment ({'HOME' => '/home/myhome'})
end
```

Delete a repository using yum to scrub the cache

```
# the following code sample thanks to gaffneyc @ https://gist.github.com/918711

execute "clean-yum-cache" do
  command "yum clean all"
  action :nothing
end

file "/etc/yum.repos.d/bad.repo" do
  action :delete
  notifies :run, "execute[clean-yum-cache]", :immediately
  notifies :create, "ruby_block[reload-internal-yum-cache]", :immediately
end
```

Install repositories from a file, trigger a command, and force the internal cache to reload

The following example shows how to install new Yum repositories from a file, where the installation of the repository triggers a creation of the Yum cache that forces the internal cache for the chef-client to reload:

```
execute "create-yum-cache" do
 command "yum -q makecache"
 action :nothing
end
```

```
ruby_block "reload-internal-yum-cache" do
  block do
    Chef::Provider::Package::Yum::YumCache.instance.reload
  end
  action :nothing
end

cookbook_file "/etc/yum.repos.d/custom.repo" do
  source "custom"
  mode '0644'
  notifies :run, "execute[create-yum-cache]", :immediately
  notifies :create, "ruby_block[reload-internal-yum-cache]", :immediately
end
```

Prevent restart and reconfigure if configuration is broken

Use the `:nothing` common action to prevent an application from restarting, and then use the `subscribes` notification to ask the broken configuration to be reconfigured immediately:

```
execute "test-nagios-config" do
  command "nagios3 --verify-config"
  action :nothing
  subscribes :run, "template[/etc/nagios3/configures-nagios.conf]", :immediately
end
```

Notify in a specific order

To notify multiple resources, and then have these resources run in a certain order, do something like the following:

```
execute 'foo' do
  command '...'
  notifies :run, 'template[baz]', :immediately
  notifies :install, 'package[bar]', :immediately
  notifies :run, 'execute[final]', :immediately
end

template 'baz' do
  ...
  notifies :run, 'execute[restart_baz]', :immediately
end

package 'bar'

execute 'restart_baz'

execute 'final' do
  command '...'
end
```

where the sequencing will be in the same order as the resources are listed in the recipe: `execute 'foo'`, `template 'baz'`, `execute [restart_baz]`, `package 'bar'`, and `execute 'final'`.

Execute a command using a template

The following example shows how to set up IPv4 packet forwarding using the **execute** resource to run a command named `forward_ipv4` that uses a template defined by the **template** resource:

```
execute "forward_ipv4" do
  command "echo > /proc/.../ipv4/ip_forward"
  action :nothing
end

template "/etc/file_name.conf" do
  source "routing/file_name.conf.erb"
  notifies :run, 'execute[forward_ipv4]', :delayed
end
```

where the `command` attribute for the **execute** resource contains the command that is to be run and the `source` attribute for the **template** resource specifies which template to use. The `notifies` attribute for the **template** specifies that the `execute[forward_ipv4]` (which is defined by the **execute** resource) should be queued up and run at the end of the chef-client run.

Add a rule to an IP table

The following example shows how to add a rule named `test_rule` to an IP table using the **execute** resource to run a command using a template that is defined by the **template** resource:

```
execute 'test_rule' do
  command "command_to_run
    --option value
    ...
    --option value
    --source #{node[:name_of_node][:ipsec][:local][:subnet]}
    -j test_rule"
  action :nothing
end

template "/etc/file_name.local" do
  source "routing/file_name.local.erb"
  notifies :run, 'execute[test_rule]', :delayed
end
```

where the `command` attribute for the **execute** resource contains the command that is to be run and the `source` attribute for the **template** resource specifies which template to use. The `notifies` attribute for the **template** specifies that the `execute[test_rule]` (which is defined by the **execute** resource) should be queued up and run at the end of the chef-client run.

Stop a service, do stuff, and then restart it

The following example shows how to use the **execute**, **service**, and **mount** resources together to ensure that a node running on Amazon EC2 is running MySQL. This example does the following:

- Checks to see if the Amazon EC2 node has MySQL
- If the node has MySQL, stops MySQL
- Installs MySQL
- Mounts the node
- Restarts MySQL

```
#  the following code sample comes from the ``server_ec2`` recipe in the following cookbook: https://github.

if (node.attribute?('ec2') && ! FileTest.directory?(node['mysql']['ec2_path']))

  service "mysql" do
    action :stop
  end

  execute "install-mysql" do
    command "mv #{node['mysql']['data_dir']} #{node['mysql']['ec2_path']}"
    not_if do FileTest.directory?(node['mysql']['ec2_path']) end
  end

  [node['mysql']['ec2_path'], node['mysql']['data_dir']].each do |dir|
    directory dir do
      owner 'mysql'
      group 'mysql'
    end
  end

  mount node['mysql']['data_dir'] do
    device node['mysql']['ec2_path']
    fstype "none"
    options "bind,rw"
    action [:mount, :enable]
  end

  service "mysql" do
    action :start
  end

end
```

where

- the two **service** resources are used to stop, and then restart the MySQL service
- the **execute** resource is used to install MySQL
- the **mount** resource is used to mount the node and enable MySQL

Use the platform_family? method

The following is an example of using the `platform_family?` method in the Recipe DSL to create a variable that can be used with other resources in the same recipe. In this example, `platform_family?` is being used to ensure that a specific binary is used for a specific platform before using the **remote_file** resource to download a file from a remote location, and then using the **execute** resource to install that file by running a command.

```
if platform_family?("rhel")
  pip_binary = "/usr/bin/pip"
else
  pip_binary = "/usr/local/bin/pip"
end

remote_file "#{Chef::Config[:file_cache_path]}/distribute_setup.py" do
  source "http://python-distribute.org/distribute_setup.py"
  mode '0644'
  not_if { ::File.exists?(pip_binary) }
end

execute "install-pip" do
  cwd Chef::Config[:file_cache_path]
  command <<-EOF
    # command for installing Python goes here
    EOF
  not_if { ::File.exists?(pip_binary) }
end
```

where a command for installing Python might look something like:

```
#{node['python']['binary']} distribute_setup.py
#{::File.dirname(pip_binary)}/easy_install pip
```

Control a service using the execute resource

Warning

This is an example of something that should NOT be done. Use the **service** resource to control a service, not the **execute** resource.

Do something like this:

```
service "tomcat" do
  action :start
end
```

and NOT something like this:

```
execute "start-tomcat" do
  command "/etc/init.d/tomcat6 start"
  action :run
end
```

There is no reason to use the **execute** resource to control a service because the **service** resource exposes the `start_command` attribute directly, which gives a recipe full control over the command issued in a much cleaner, more direct manner.

Use the search recipe DSL method to find users

The following example shows how to use the `search` method in the Recipe DSL to search for users:

```
#  the following code sample comes from the openvpn cookbook: https://github.com/opscode-cookbooks/openvpn

search("users", "*:*") do |u|
  execute "generate-openvpn-#{u['id']}" do
    command "./pkitool #{u['id']}"
    cwd "/etc/openvpn/easy-rsa"
    environment(
      'EASY_RSA' => '/etc/openvpn/easy-rsa',
      'KEY_CONFIG' => '/etc/openvpn/easy-rsa/openssl.cnf',
      'KEY_DIR' => node["openvpn"]["key_dir"],
      'CA_EXPIRE' => node["openvpn"]["key"]["ca_expire"].to_s,
      'KEY_EXPIRE' => node["openvpn"]["key"]["expire"].to_s,
      'KEY_SIZE' => node["openvpn"]["key"]["size"].to_s,
      'KEY_COUNTRY' => node["openvpn"]["key"]["country"],
      'KEY_PROVINCE' => node["openvpn"]["key"]["province"],
      'KEY_CITY' => node["openvpn"]["key"]["city"],
      'KEY_ORG' => node["openvpn"]["key"]["org"],
      'KEY_EMAIL' => node["openvpn"]["key"]["email"]
    )
    not_if { ::File.exists?("#{node["openvpn"]["key_dir"]}/#{u['id']}.crt") }
  end

  %w{ conf ovpn }.each do |ext|
    template "#{node["openvpn"]["key_dir"]}/#{u['id']}.#{ext}" do
      source "client.conf.erb"
      variables :username => u['id']
    end
  end

  execute "create-openvpn-tar-#{u['id']}" do
    cwd node["openvpn"]["key_dir"]
    command <<-EOH
      tar zcf #{u['id']}.tar.gz \
      ca.crt #{u['id']}.crt #{u['id']}.key \
      #{u['id']}.conf #{u['id']}.ovpn \
    EOH
    not_if { ::File.exists?("#{node["openvpn"]["key_dir"]}/#{u['id']}.tar.gz") }
  end
end
```

where

- the search will use both of the **execute** resources, unless the condition specified by the `not_if` commands are met
- the `environments` attribute in the first **execute** resource is being used to define values that appear as variables in the OpenVPN configuration
- the **template** resource tells the chef-client which template to use

Enable remote login for Mac OS X

```
execute "enable ssh" do
  command "/usr/sbin/systemsetup -setremotelogin on"
  not_if "/usr/sbin/systemsetup -getremotelogin | /usr/bin/grep On"
  action :run
end
```

Execute code immediately, based on the template resource

By default, notifications are `:delayed`, that is they are queued up as they are triggered, and then executed at the very end of a chef-client run. To run an action immediately, use `:immediately`:

```
template "/etc/nagios3/configures-nagios.conf" do
  # other parameters
  notifies :run, "execute[test-nagios-config]", :immediately
end
```

and then the chef-client would immediately run the following:

```
execute "test-nagios-config" do
  command "nagios3 --verify-config"
  action :nothing
end
```

Run a Knife command

```
execute 'create_user' do
  command <<-EOM.gsub(/\s+/, ' ').strip!
        knife user create #{user}
      --admin
      --password password
      --disable-editing
      --file /home/vagrant/.chef/user.pem
      --config /tmp/knife-admin.rb
    EOM
end
```

file

Use the **file** resource to manage files directly on a node.

Note

Use the **cookbook_file** resource to copy a file from a cookbook's `/files` directory. Use the **template** resource to create a file based on a template in a cookbook's `/templates` directory. And use the **remote_file** resource to transfer a file to a node from a remote location.

Syntax

The syntax for using the **file** resource in a recipe is as follows:

```
file "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `file` tells the chef-client to use the `Chef::Provider::File` provider during the chef-client run
- `name` is the name of the resource block; when the `path` attribute is not specified as part of a recipe, `name` is also the path to the file
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
file "/tmp/something" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

Actions

This resource has the following actions:

Action	Description

Action	Description
`:create`	Default. Use to create a file. If a file already exists (but does not match), use to update that file to match.
`:create_if_missing`	Use to create a file only if the file does not exist. (When the file exists, nothing happens.)
`:delete`	Use to delete a file.
`:touch`	Use to touch a file. This updates the access (atime) and file modification (mtime) times for a file.

Attributes

This resource has the following attributes:

Attribute	Description
`atomic_update`	Use to perform atomic file updates on a per-resource basis. Set to `true` for atomic file updates. Set to `false` for non-atomic file updates. (This setting overrides `file_atomic_update`, which is a global setting found in the client.rb file.) Default value: `true`.
`backup`	The number of backups to be kept. Set to `false` to prevent backups from being kept. Default value: `5`.
`checksum`	The SHA-256 checksum of the file. Use this attribute to ensure that a specific file is used. If the checksum does not match, the file will not be used. Default value: no checksum required.
`content`	A string that is written to the file. The contents of this attribute will replace any previous content when this attribute has something other than the default value. The default behavior will not modify content.
`force_unlink`	Use to specify how the chef-client handles certain situations when the target file turns out not to be a file. For example, when a target file is actually a symlink. Set to `true` to have the chef-client delete the non-file target and replace it with the specified file. Set to `false` for the chef-client to raise an error. Default value: `false`.
`group`	A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`manage_symlink_source`	Use to have the chef-client detect and manage the source file for a symlink. Possible values: `nil`, `true`, or `false`. When this value is set to `nil`, the chef-client will manage a symlink's source file and emit a warning. When this value is set to `true`, the chef-client will manage a symlink's source file and not emit a warning. Default value: `nil`. The default value will be changed to `false` in a future version.
`mode`	A quoted string that defines the octal mode for a file. If `mode` is not specified and if the file already exists, the existing mode on the file is used. If `mode` is not specified, the file does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the file will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`owner`	A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`path`	The path to the file. Using a fully qualified path is recommended, but is not always required. Default value: the `name` of the resource block. (See "Syntax" section above for more information.) Microsoft Windows: A path that begins with a forward slash (`/`) will point to the root of the current working directory of the chef-client process. This path can vary from system to system. Therefore, using

Attribute	Description
	a path that begins with a forward slash (`/`) is not recommended.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`rights`	Microsoft Windows only. The permissions for users and groups in a Microsoft Windows environment. For example: `rights <permissions>, <principal>, <options>` where `<permissions>` specifies the rights granted to the principal, `<principal>` is the group or user name, and `<options>` is a Hash with one (or more) advanced rights options.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::File`	`file`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create a file

```
file "/tmp/something" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

Create a file in Microsoft Windows

```
file "C:\tmp\something.txt" do
  rights :read, "Everyone"
  rights :full_control, "DOMAIN\User"
  action :create
end
```

Remove a file

```
file "/tmp/something" do
  action :delete
end
```

Set file modes

```
file "/tmp/something" do
  mode '0644'
end
```

Delete a repository using yum to scrub the cache

```
# the following code sample thanks to gaffneyc @ https://gist.github.com/918711

execute "clean-yum-cache" do
  command "yum clean all"
  action :nothing
end

file "/etc/yum.repos.d/bad.repo" do
  action :delete
  notifies :run, "execute[clean-yum-cache]", :immediately
  notifies :create, "ruby_block[reload-internal-yum-cache]", :immediately
end
```

Add the value of a data bag item to a file

The following example shows how to get the contents of a data bag item named `impossible_things`, create a .pem file located at `some/directory/path/`, and then use the `content` attribute to update the contents of that file with the value of the `impossible_things` data bag item:

```
private_key = data_bag_item("impossible_things", private_key_name)["private_key"]

file "some/directory/path/#{private_key_name}.pem" do
  content private_key
  owner 'root'
  group 'group'
  mode '640'
end
```

Write a YAML file

The following example shows how to use the `content` attribute to write a YAML file:

```
file "#{app['deploy_to']}/shared/config/settings.yml" do
  owner 'app["owner"]'
  group 'app["group"]'
  mode '644'
  content app.to_yaml
end
```

Write a string to a file

The following example specifies a directory, and then uses the `content` attribute to add a string to the file created in that directory:

```
status_file = "/path/to/file/status_file"

file status_file do
  owner 'root'
  group 'root'
  mode '0600'
  content "My favourite foremost coastal Antarctic shelf, oh Larsen B!"
end
```

Create a file from a copy

The following example shows how to copy a file from one directory to another, locally on a node:

```
file "/root/1.txt" do
  content IO.read("/tmp/1.txt")
  action :create
end
```

where the `content` attribute uses the Ruby `IO.read` method to get the contents of the `/tmp/1.txt` file.

freebsd_package

Use the **freebsd_package** resource to manage packages for the FreeBSD platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **freebsd_package** resource in a recipe is as follows:

```
freebsd_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `freebsd_package` tells the chef-client to use the `Chef::Provider::Package::Freebsd` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.

Action	Description
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Freebsd`	`freebsd_package`	The provider that is used with the FreeBSD platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
freebsd_package "name of package" do
  action :install
end
```

gem_package

Use the **gem_package** resource to manage gem packages that are only included in recipes. When a package is installed from a local file, it must be added to the node using the **remote_file** or **cookbook_file** resources.

Warning

The **chef_gem** and **gem_package** resources are both used to install Ruby gems. For any machine on which the chef-client is installed, there are two instances of Ruby. One is the standard, system-wide instance of Ruby and the other is a dedicated instance that is available only to the chef-client. Use the **chef_gem** resource to install gems into the instance of Ruby that is dedicated to the chef-client. Use the **gem_package** resource to install all other gems (i.e. install gems system-wide).

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **gem_package** resource in a recipe is as follows:

```
gem_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `gem_package` tells the chef-client to use the `Chef::Provider::Package::Rubygems` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Gem Package Options

The RubyGems package provider attempts to use the RubyGems API to install gems without spawning a new process, whenever possible. A gems command to install will be spawned under the following conditions:

- When a `gem_binary` attribute is specified (as a hash, a string, or by a .gemrc file), the provider will run that command to examine its environment settings and then again to install the gem.
- When install options are specified as a string, the provider will span a gems command with those options when installing the gem.
- The omnibus installer will search the PATH for a gem command rather than defaulting to the current gem environment. As part of `enforce_path_sanity`, the `bin` directories area added to the PATH, which means when there are no other proceeding RubyGems, the installation will still be operated against it.

Use a Hash

If an explicit `gem_binary` parameter is not being used with the `gem_package` resource, it is preferable to provide the install options as a hash. This approach allows the provider to install the gem without needing to spawn an external gem process.

The following RubyGems options are available for inclusion within a hash and are passed to the RubyGems DependencyInstaller:

- `:env_shebang`
- `:force`
- `:format_executable`
- `:ignore_dependencies`
- `:prerelease`
- `:security_policy`
- `:wrappers`

For more information about these options, see the RubyGems documentation: http://rubygems.rubyforge.org/rubygems-update/Gem/DependencyInstaller.html.

Example

```
gem_package "bundler" do
  options(:prerelease => true, :format_executable => false)
end
```

Use a String

When using an explicit `gem_binary`, options must be passed as a string. When not using an explicit `gem_binary`, the chef-client is forced to spawn a gems process to install the gems (which uses more system resources) when options are passed as a string. String options are passed verbatim to the gems command and should be specified just as if they were passed on a command line. For example, `--prerelease` for a pre-release gem.

Example

```
gem_package "nokogiri" do
  gem_binary("/opt/ree/bin/gem")
  options("--prerelease --no-format-executable")
end
```

Use a .gemrc File

Options can be specified in a .gemrc file. By default the `gem_package` resource will use the Ruby interface to install gems which will ignore the .gemrc file. The `gem_package` resource can be forced to use the gems command instead (and to read the .gemrc file) by adding the `gem_binary` attribute to a code block.

Example

```
gem_package "nokogiri" do
  gem_binary "gem"
end
```

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:reconfig`	Use to reconfigure a package. This action requires a response file.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`gem_binary`	An attribute for the `gem_package` provider that is used to specify a gems binary. By default, the same version of Ruby that is used by the chef-client will be installed.
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The URL at which the gem package is located.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Rubygems`	`gem_package`	Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a gems file from the local file system

```
gem_package "right_aws" do
  source "/tmp/right_aws-1.11.0.gem"
  action :install
end
```

Use the ignore_failure common attribute

```
gem_package "syntax" do
  action :install
  ignore_failure true
end
```

git

Use the **git** resource to manage source control resources that exist in a git repository. git version 1.6.5 (or higher) is required to use all of the functionality in the **git** resource.

Note

This resource is often used in conjunction with the **deploy** resource.

Syntax

The syntax for using the **git** resource in a recipe is as follows:

```
git "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `git` tells the chef-client to use the `Chef::Provider::Git` provider during the chef-client run.
- `"name"` is the location in which the source files will be placed and/or synchronized with the files under source control management
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
git "#{Chef::Config[:file_cache_path]}/app_name" do
  repository node[:app_name][:git_repository]
  revision node[:app_name][:git_revision]
  action :sync
  notifies :run, "bash[compile_app_name]"
end
```

where

- the name of the resource is `#{Chef::Config[:file_cache_path]}/libvpx`
- the `repository` and `reference` nodes tell the chef-client which repository and revision to use

Actions

This resource has the following actions:

Action	Description
`:sync`	Default. Use to update the source to the specified version, or to get a new clone or checkout.
`:checkout`	Use to clone or check out the source. When a checkout is available, this provider does nothing.
`:export`	Use to export the source, excluding or removing any version control artifacts.

Attributes

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Git`	`git`	This provider works only with git.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Use the git mirror

```
git "/opt/mysources/couch" do
  repository "git://git.apache.org/couchdb.git"
  revision "master"
  action :sync
end
```

Use different branches

To use different branches, depending on the environment of the node:

```
if node.chef_environment == "QA"
   branch_name = "staging"
else
   branch_name = "master"
end

git "/home/user/deployment" do
   repository "git@github.com:gitsite/deployment.git"
   revision branch_name
   action :sync
```

```
    user "user"
    group "test"
 end
```

where the `branch_name` variable is set to `staging` or `master`, depending on the environment of the node. Once this is determined, the `branch_name` variable is used to set the revision for the repository. If the `git status` command is used after running the example above, it will return the branch name as `deploy`, as this is the default value. Run the chef-client in debug mode to verify that the correct branches are being checked out:

```
$ sudo chef-client -l debug
```

Install an application from git using bash

The following example shows how Bash can be used to install a plug-in for rbenv named `ruby-build`, which is located in git version source control. First, the application is synchronized, and then Bash changes its working directory to the location in which `ruby-build` is located, and then runs a command.

```
  git "#{Chef::Config[:file_cache_path]}/ruby-build" do
    repository "git://github.com/sstephenson/ruby-build.git"
    reference "master"
    action :sync
  end

  bash "install_ruby_build" do
    cwd "#{Chef::Config[:file_cache_path]}/ruby-build"
    user "rbenv"
    group "rbenv"
    code <<-EOH
      ./install.sh
      EOH
    environment 'PREFIX' => "/usr/local"
 end
```

To read more about `ruby-build`, see here: https://github.com/sstephenson/ruby-build.

Upgrade packages from git

The following example shows the **scm** resource using the `git` short name as part of a larger recipe that is used to upgrade packages:

```
#  the following code sample comes from the ``source`` recipe in the ``libvpx-cookbook`` cookbook: https://g

git "#{Chef::Config[:file_cache_path]}/libvpx" do
  repository node[:libvpx][:git_repository]
  revision node[:libvpx][:git_revision]
  action :sync
  notifies :run, "bash[compile_libvpx]", :immediately
end
```

group

Use the **group** resource to manage a local group.

Syntax

The syntax for using the **group** resource in a recipe is as follows:

```
group "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `group` tells the chef-client to use one of the following providers during the chef-client run: `Chef::Provider::Group`, `Chef::Provider::Group::Aix`, `Chef::Provider::Group::Dscl`, `Chef::Provider::Group::Gpasswd`, `Chef::Provider::Group::Groupadd`, `Chef::Provider::Group::Groupmod`, `Chef::Provider::Group::Pw`, `Chef::Provider::Group::Suse`, `Chef::Provider::Group::Usermod`, or `Chef::Provider::Group::Windows`. The provider that is used by the chef-client depends on the platform of the machine on which the chef-client run is taking place
- `name` is the name of the resource block; when the `group_name` attribute is not specified as part of a recipe, `name` is also the name of the group
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a group. If a group already exists (but does not match), use to update that group to match.
`:remove`	Use to remove a group.
`:modify`	Use to modify an existing group. This action will raise an exception if the group does not exist.
`:manage`	Use to manage an existing group. This action will do nothing if the group does not exist.

Attributes

This resource has the following attributes:

Attribute	Description
`append`	Use to specify how members should be appended and/or removed from a group. When `true`, `members` will be appended and `excluded_members` will be removed. When `false`, group members will be reset to the value of the `members` attribute. Default value: `false`.
`excluded_members`	Use to remove users from a group. May only be used when `append` is set to `true`.
`gid`	The identifier for the group.
`group_name`	The name of the group. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`members`	Use to specify which users should be set or appended to a group. When more than one group member is identified, the list of members should be an array: `members ['user1', 'user2']`.
`non_unique`	Use to allow `gid` duplication. May only be used with the `Groupadd` provider. Default value: `false`.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`system`	Use to show if a group belongs to a system group. Set to `true` if the group belongs to a system group.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Group`	group	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Group::Aix`	group	The provider that is used with the AIX platform.
`Chef::Provider::Group::Dscl`	group	The provider that is used with the Mac OS X platform.
`Chef::Provider::Group::Gpasswd`	group	The provider that is used with the gpasswd command.
`Chef::Provider::Group::Groupadd`	group	The provider that is used with the groupadd command.
`Chef::Provider::Group::Groupmod`	group	The provider that is used with the groupmod command.
`Chef::Provider::Group::Pw`	group	The provider that is used with the FreeBSD platform.
`Chef::Provider::Group::Suse`	group	The provider that is used with the openSUSE platform.
`Chef::Provider::Group::Usermod`	group	The provider that is used with the Solaris platform.
`Chef::Provider::Group::Windows`	group	The provider that is used with the Microsoft Windows platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Append users to groups

```
group "www-data" do
  action :modify
  members "maintenance"
  append true
end
```

http_request

Use the **http_request** resource to send an HTTP request (GET, PUT, POST, DELETE, HEAD, or OPTIONS) with an arbitrary message. This resource is often useful when custom callbacks are necessary.

Syntax

The syntax for using the **http_request** resource in a recipe is as follows:

```
http_request "name" do
  url "http://some.url"
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `http_request` tells the chef-client to use the `Chef::Provider::HttpRequest` provider during the chef-client run
- `name` is the name of the resource block; when the `message` attribute is not specified as part of a recipe, `name` is also the message that is sent by the HTTP request
- `attribute` is zero (or more) of the attributes that are available for this resource
- `url` is the URL that will precede `?message=` in the HTTP request
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example, send a `DELETE` request to "http://www.getchef.com/some_page?message=please_delete_me".

```
http_request "please_delete_me" do
  url "http://www.getchef.com/some_page"
  action :delete
end
```

Actions

This resource has the following actions:

Action	Description
`:get`	Default. Use to send a `GET` request.
`:put`	Use to send a `PUT` request.
`:post`	Use to send a `POST` request.
`:delete`	Use to send a `DELETE` request.
`:head`	Use to send a `HEAD` request.
`:options`	Use to send an `OPTIONS` request.

Attributes

This resource has the following attributes:

Attribute	Description
`headers`	A Hash of custom headers. Default value: `{}`.
`message`	The message that is sent by the HTTP request. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`url`	The URL to which an HTTP request is sent.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::HttpRequest`	`http_request`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Send a GET request

```
http_request "some_message" do
  url "http://example.com/check_in"
end
```

The message is sent as "http://example.com/check_in?message=some_message".

Send a POST request

To send a `POST` request as JSON data, convert the message to JSON and include the correct content-type header. For example:

```
http_request "posting data" do
  action :post
  url "http://example.com/check_in"
  message ({:some => "data"}.to_json)
  headers({"AUTHORIZATION" => "Basic #{Base64.encode64("username:password")}","Content-Type" => "application
end
```

Transfer a file only when the remote source changes

```
remote_file "/tmp/couch.png" do
  source "http://couchdb.apache.org/img/sketch.png"
  action :nothing
end

http_request "HEAD http://couchdb.apache.org/img/sketch.png" do
  message ""
  url "http://couchdb.apache.org/img/sketch.png"
  action :head
  if File.exists?("/tmp/couch.png")
    headers "If-Modified-Since" => File.mtime("/tmp/couch.png").httpdate
  end
  notifies :create, "remote_file[/tmp/couch.png]", :immediately
end
```

ifconfig

Use the **ifconfig** resource to manage interfaces.

Syntax

The syntax for using the **ifconfig** resource in a recipe is as follows:

```
ifconfig "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `ifconfig` tells the chef-client to use the `Chef::Provider::Ifconfig` provider during the chef-client run
- `name` is the name of the resource block; when the `target` attribute is not specified as part of a recipe, `name` is also the IP address that will be assigned to the network interface
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:add`	Default. Use to run ifconfig to configure a network interface and (on some platforms) write a configuration file for that network interface.
`:delete`	Use to run ifconfig to disable a network interface and (on some platforms) delete that network interface's configuration file.
`:enable`	Use to run ifconfig to enable a network interface.
`:disable`	Use to run ifconfig to disable a network interface.

Attributes

This resource has the following attributes:

Attribute	Description
`bcast`	The broadcast address for a network interface. One some platforms this attribute is not set using ifconfig, but is instead added to the startup configuration file for the network interface.
`bootproto`	The boot protocol used by a network interface.
`device`	The network interface to be configured.
`hwaddr`	The hardware address for the network interface.
`inet_addr`	The Internet host address for the network interface.
`mask`	The decimal representation of the network mask. For example: `255.255.255.0`.
`metric`	The routing metric for the interface.
`mtu`	The maximum transmission unit (MTU) for the network interface.
`network`	The address for the network interface.
`onboot`	Use to bring up the network interface on boot.
`onparent`	Use to bring up the network interface when its parent interface is brought up.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`target`	The IP address that will be assigned to the network interface. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Ifconfig`	`ifconfig`	The default provider for all platforms. Currently, this provider only writes out a start-up configuration file for the interface on Red Hat-based platforms (it writes to `/etc/sysconfig/network-scripts/ifcfg-#{device_name}`).

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Configure a network interface

```
ifconfig "192.186.0.1" do
  device "eth0"
end
```

ips_package

Use the **ips_package** resource to manage packages (using Image Packaging System (IPS)) on the Solaris 11 platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **ips_package** resource in a recipe is as follows:

```
ips_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `ips_package` tells the chef-client to use the `Chef::Provider::Package::Ips` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`accept_license`	Use to accept an end-user license agreement automatically. Default value: `false`.
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Ips`	`ips_package`	The provider that is used with the ips platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
ips_package "name of package" do
  action :install
end
```

link

Use the **link** resource to create symbolic or hard links.

A symbolic link—sometimes referred to as a soft link—is a directory entry that associates a file name with a string that contains an absolute or relative path to a file on any file system. In other words, "a file that contains a path that points to another file." A symbolic link creates a new file with a new inode that points to the inode location of the original file.

A hard link is a directory entry that associates a file with another file in the same file system. In other words, "multiple directory entries to the same file." A hard link creates a new file that points to the same inode as the original file.

Syntax

The syntax for using the **link** resource in a recipe is as follows:

```
link "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `link` tells the chef-client to use the `Chef::Provider::Link` provider during the chef-client run
- `name` is the name of the resource block; when the `target_file` attribute is not specified as part of a recipe, `name` is also name of the link
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a link. If a link already exists (but does not match), use to update that link to match.
`:delete`	Use to delete a link.

Attributes

This resource has the following attributes:

Attribute	Description
`group`	A string or ID that identifies the group associated with a symbolic link.
`link_type`	The type of link: `:symbolic` or `:hard`. Default value: `:symbolic`.
`owner`	The owner associated with a symbolic link.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`target_file`	The name of the link. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`to`	The actual file to which the link will be created.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Link`	`link`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create symbolic links

The following example will create a symbolic link from `/tmp/file` to `/etc/file`:

```
link "/tmp/file" do
  to "/etc/file"
end
```

Create hard links

The following example will create a hard link from `/tmp/file` to `/etc/file`:

```
link "/tmp/file" do
  to "/etc/file"
  link_type :hard
end
```

Delete links

The following example will delete the `/tmp/file` symbolic link and uses the `only_if` guard to run the `test -L` command, which verifies that `/tmp/file` is a symbolic link, and then only deletes `/tmp/file` if the test passes:

```
link "/tmp/file" do
  action :delete
  only_if "test -L /tmp/file"
end
```

Create multiple symbolic links

The following example creates symbolic links from two files in the `/vol/webserver/cert/` directory to files located in the `/etc/ssl/certs/` directory:

```
link "/vol/webserver/cert/server.crt" do
  to "/etc/ssl/certs/ssl-cert-name.pem"
end

link "/vol/webserver/cert/server.key" do
  to "/etc/ssl/certs/ssl-cert-name.key"
end
```

Create platform-specific symbolic links

The following example shows installing a filter module on Apache. The package name is different for different platforms, and for the Red Hat Enterprise Linux family, a symbolic link is required:

```
include_recipe 'apache2::default'

case node['platform_family']
when 'debian'
  ...
when 'suse'
  ...
when 'rhel', 'fedora'
  ...


  link '/usr/lib64/httpd/modules/mod_apreq.so' do
    to      '/usr/lib64/httpd/modules/mod_apreq2.so'
    only_if 'test -f /usr/lib64/httpd/modules/mod_apreq2.so'
  end

  link '/usr/lib/httpd/modules/mod_apreq.so' do
    to      '/usr/lib/httpd/modules/mod_apreq2.so'
    only_if 'test -f /usr/lib/httpd/modules/mod_apreq2.so'
  end
end

...
```

For the entire recipe, see https://github.com/onehealth-cookbooks/apache2/blob/68bdfba4680e70b3e90f77e40223dd535bf22c17/recipes/mod_apreq2.rb.

log

Use the **log** resource to to create log entries from a recipe.

Syntax

The syntax for using the **log** resource in a recipe is as follows:

```
log "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `log` tells the chef-client to use the `Chef::Provider::Log::ChefLog` provider during the chef-client run
- `name` is the name of the resource block; when the `message` attribute is not specified as part of a recipe, `name` is also the message to be added to a log file
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description

Action	Description
`:write`	Default. Use to write to log.

Attributes

This resource has the following attributes:

Attribute	Description
`level`	The level of logging that will be displayed by the chef-client. The chef-client uses the `mixlib-log` (https://github.com/opscode/mixlib-log) to handle logging behavior. Options (in order of priority): `:debug`, `:info`, `:warn`, `:error`, and `:fatal`. Default value: `:info`.
`message`	The message to be added to a log file. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Log::ChefLog`	`log`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Set default logging level

```
log "your string to log"
```

Set debug logging level

```
log "a debug string" do
  level :debug
end
```

Add a message to a log file

```
log "message" do
  message "This is the message that will be added to the log."
  level :info
end
```

macports_package

Use the **macports_package** resource to manage packages for the Mac OS X platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **macports_package** resource in a recipe is as follows:

```
macports_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `macports_package` tells the chef-client to use the `Chef::Provider::Package::Macports` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package

- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Macports`	`macports_package`	The provider that is used with the Mac OS X platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
macports_package "name of package" do
  action :install
end
```

mdadm

Use the **mdadm** resource to manage RAID devices in a Linux environment using the mdadm utility. The **mdadm** provider will create and assemble an array, but it will not create the config file that is used to persist the array upon reboot. If the config file is required, it must be done by specifying a template with the correct array layout, and then by using the **mount** provider to create a file systems table (fstab) entry.

Syntax

The syntax for using the **mdadm** resource in a recipe is as follows:

```
mdadm "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `mdadm` tells the chef-client to use the `Chef::Provider::Mdadm` provider during the chef-client run
- `name` is the name of the resource block; when the `raid_device` attribute is not specified as part of a recipe, `name` is also the name of the RAID device
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create an array with per-device superblocks. If an array already exists (but does not match), use to update that array to match.
`:assemble`	Use to assemble a previously created array into an active array.
`:stop`	Use to stop an active array.

Attributes

This resource has the following attributes:

Attribute	Description
`bitmap`	The path to a file in which a write-intent bitmap is stored.
`chunk`	The chunk size. This attribute should not be used for a RAID 1 mirrored pair (i.e. when the `level` attribute is set to `1`). Default value: `16`.
`devices`	A comma-separated list of devices to be part of a RAID array. Default value: `[]`.
`exists`	Indicates whether the RAID array exists. Default value: `false`.
`level`	The RAID level. Default value: `1`.
`metadata`	The superblock type for RAID metadata. Default value: `0.90`.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`raid_device`	The name of the RAID device. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Mdadm`	`mdadm`	The default provider for the Linux platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create and assemble a RAID 0 array

The mdadm command can be used to create RAID arrays. For example, a RAID 0 array named `/dev/md0` with 10 devices would have a command similar to the following:

```
$ mdadm --create /dev/md0 --level=0 --raid-devices=10 /dev/s01.../dev/s10
```

where `/dev/s01 .. /dev/s10` represents 10 devices (01, 02, 03, and so on). This same command, when expressed as a recipe using the **mdadm** resource, would be similar to:

```
mdadm "/dev/md0" do
  devices [ "/dev/s01", ... "/dev/s10" ]
  level 0
  action :create
end
```

(again, where `/dev/s01 .. /dev/s10` represents devices /dev/s01, /dev/s02, /dev/s03, and so on).

Create and assemble a RAID 1 array

```
mdadm "/dev/md0" do
  devices [ "/dev/sda", "/dev/sdb" ]
  level 1
  action [ :create, :assemble ]
end
```

Create and assemble a RAID 5 array

The mdadm command can be used to create RAID arrays. For example, a RAID 5 array named `/dev/sd0` with 4, and a superblock type of `0.90` would be similar to:

```
mdadm "/dev/sd0" do
  devices [ "/dev/s1", "/dev/s2", "/dev/s3", "/dev/s4" ]
  level 5
  metadata "0.90"
  chunk 32
  action :create
end
```

mount

Use the **mount** resource to manage a mounted file system.

Syntax

The syntax for using the **mount** resource in a recipe is as follows:

```
mount "name" do
  attribute "value" # see attributes section below
  ...
  fstype "type"
  action :action # see actions section below
end
```

where

- `mount` tells the chef-client to use the `Chef::Provider::Mount` provider during the chef-client run for all platforms except for Microsoft Windows, which uses the `Chef::Provider::Mount::Windows` provider
- `name` is the name of the resource block; when the `mount_point` attribute is not specified as part of a recipe, `name` is also the directory (or path) in which a device should be mounted
- `attribute` is zero (or more) of the attributes that are available for this resource
- `fstype` is the file system type; this attribute is required
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
mount node['mysql']['ec2_path'] do
  device ebs_vol_dev
  fstype "xfs"
  action :mount
end
```

Actions

This resource has the following actions:

Action	Description
`:mount`	Default. Use to mount a device.
`:umount`	Use to unmount a device.
`:remount`	Use to remount a device.
`:enable`	Use to add an entry to the file systems table (fstab).
`:disable`	Use to remove an entry from the file systems table (fstab).

Note

Order matters when passing multiple actions. For example: `action [:mount, :enable]` ensures that the file system is mounted before it is enabled.

Attributes

This resource has the following attributes:

Attribute	Description
device	Required for :umount and :remount actions (for the purpose of checking the mount command output for presence). The special block device or remote node, a label, or a uuid to be mounted.
device_type	The type of device: :device, :label, or :uuid. Default value: :device.
domain	Microsoft Windows only. Use to specify the domain in which the username and password are located.
dump	The dump frequency (in days) used while creating a file systems table (fstab) entry. Default value: 0.
enabled	Use to specify if a mounted file system is enabled. Default value: false.
fstype	Required. The file system type (fstype) of the device.
mount_point	The directory (or path) in which the device should be mounted. Default value: the name of the resource block. (See “Syntax” section above for more information.)
mounted	Use to specify if a file system is already mounted. Default value: false.
options	An array or string that contains mount options. If this value is a string, it will be converted to an array. Default value: defaults.
pass	The pass number used by the file system check (fsck) command while creating a file systems table (fstab) entry. Default value: 2.
password	Microsoft Windows only. Use to specify the password for username.
provider	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
supports	A Hash of options for supported mount features. Default value: { :remount => false }.
username	Microsoft Windows only. Use to specify the user name.

Providers

This resource has the following providers:

Long name	Short name	Notes
Chef::Provider::Mount	mount	The default provider for all platforms, except for Microsoft Windows.
Chef::Provider::Mount::Windows	mount	The default provider for the Microsoft Windows platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Mount a labeled file system

```
mount "/mnt/volume1" do
  device "volume1"
  device_type :label
  fstype "xfs"
  options "rw"
end
```

Mount a local block drive

```
mount "/mnt/local" do
  device "/dev/sdb1"
  fstype "ext3"
end
```

Mount a non-block file system

```
mount "/mount/tmp" do
  pass     0
  fstype   "tmpfs"
  device   "/dev/null"
  options  "nr_inodes=999k,mode=755,size=500m"
  action   [:mount, :enable]
end
```

Mount and add to the file systems table

```
mount "/export/www" do
```

```
  device "nas1prod:/export/web_sites"
  fstype "nfs"
  options "rw"
  action [:mount, :enable]
end
```

Mount a remote file system

```
mount "/export/www" do
  device "nas1prod:/export/web_sites"
  fstype "nfs"
  options "rw"
end
```

Mount a remote folder in Microsoft Windows

```
mount "T:" do
  action :mount
  device "\\\\hostname.example.com\\folder"
end
```

Unmount a remote folder in Microsoft Windows

```
mount "T:" do
  action :umount
  device "\\\\hostname.example.com\\D$"
end
```

Stop a service, do stuff, and then restart it

The following example shows how to use the **execute**, **service**, and **mount** resources together to ensure that a node running on Amazon EC2 is running MySQL. This example does the following:

- Checks to see if the Amazon EC2 node has MySQL
- If the node has MySQL, stops MySQL
- Installs MySQL
- Mounts the node
- Restarts MySQL

```
#  the following code sample comes from the ``server_ec2`` recipe in the following cookbook: https://github.

if (node.attribute?('ec2') && ! FileTest.directory?(node['mysql']['ec2_path']))

  service "mysql" do
    action :stop
  end

  execute "install-mysql" do
    command "mv #{node['mysql']['data_dir']} #{node['mysql']['ec2_path']}"
    not_if do FileTest.directory?(node['mysql']['ec2_path']) end
  end

  [node['mysql']['ec2_path'], node['mysql']['data_dir']].each do |dir|
    directory dir do
      owner 'mysql'
      group 'mysql'
    end
  end

  mount node['mysql']['data_dir'] do
    device node['mysql']['ec2_path']
    fstype "none"
    options "bind,rw"
    action [:mount, :enable]
  end

  service "mysql" do
    action :start
  end

end
```

where

- the two **service** resources are used to stop, and then restart the MySQL service
- the **execute** resource is used to install MySQL
- the **mount** resource is used to mount the node and enable MySQL

ohai

Use the **ohai** resource to reload the Ohai configuration on a node. This allows recipes that change system attributes (like a recipe that adds a user) to refer to those attributes later on during the chef-client run.

Syntax

The syntax for using the **ohai** resource in a recipe is as follows:

```
ohai "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `ohai` tells the chef-client to use the `Chef::Provider::Ohai` provider during the chef-client run
- `"name"` is a friendly name for the action that is defined in the recipe
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:reload`	Default. Use to reload the Ohai configuration on a node.

Attributes

This resource has the following attributes:

Attribute	Description
`name`	Always the same value as the `name` of the resource block. (See “Syntax” section above for more information.)
`plugin`	Optional. The attribute to be reloaded. The chef-client will identify the correct plugin.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Ohai`	`ohai`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Reload Ohai

```
ohai "reload" do
  action :reload
end
```

Reload Ohai after a new user is created

```
ohai "reload_passwd" do
  action :nothing
  plugin "etc"
end

user "daemonuser" do
  home "/dev/null"
  shell "/sbin/nologin"
  system true
  notifies :reload, "ohai[reload_passwd]", :immediately
end

ruby_block "just an example" do
  block do
    # These variables will now have the new values
    puts node['etc']['passwd']['daemonuser']['uid']
    puts node['etc']['passwd']['daemonuser']['gid']
  end
end
```

package

Use the **package** resource to manage packages. When the package is installed from a local file (such as with RubyGems, dpkg, or RPM Package Manager), the file must be added to the node using the **remote_file** or **cookbook_file** resources.

Note

There are a number of platform-specific resources available for package management. In general, the **package** resource will use the correct package manager based on the platform-specific details collected by Ohai at the start of the chef-client run, which means that the platform-specific resources are often unnecessary. That said, there are cases when using a platform-specific package-based resource is desired. See the following resources for more information about these platform-specific resources: `apt_package`, `chef_gem`, `dpkg_package`, `easy_install_package`, `freebsd_package`, `gem_package`, `ips_package`, `macports_package`, `pacman_package`, `portage_package`, `rpm_package`, `smartos_package`, `solaris_package`, and `yum_package`.

Syntax

The syntax for using the **package** resource in a recipe is as follows:

```
package "name" do
  some_attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `package` tells the chef-client to use one of sixteen different providers during the chef-client run, where the provider that is used by chef-client depends on the platform of the machine on which the chef-client run is taking place
- `"name"` is the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Gem Package Options

The RubyGems package provider attempts to use the RubyGems API to install gems without spawning a new process, whenever possible. A gems command to install will be spawned under the following conditions:

- When a `gem_binary` attribute is specified (as a hash, a string, or by a .gemrc file), the provider will run that command to examine its environment settings and then again to install the gem.
- When install options are specified as a string, the provider will span a gems command with those options when installing the gem.
- The omnibus installer will search the PATH for a gem command rather than defaulting to the current gem environment. As part of `enforce_path_sanity`, the `bin` directories area added to the PATH, which means when there are no other proceeding RubyGems, the installation will still be operated against it.

Specify Options with a Hash

If an explicit `gem_binary` parameter is not being used with the `gem_package` resource, it is preferable to provide the install options as a hash. This approach allows the provider to install the gem without needing to spawn an external gem process.

The following RubyGems options are available for inclusion within a hash and are passed to the RubyGems DependencyInstaller:

- `:env_shebang`
- `:force`
- `:format_executable`
- `:ignore_dependencies`
- `:prerelease`
- `:security_policy`
- `:wrappers`

For more information about these options, see the RubyGems documentation: http://rubygems.rubyforge.org/rubygems-update/Gem/DependencyInstaller.html.

Example

```
gem_package "bundler" do
  options(:prerelease => true, :format_executable => false)
end
```

Specify Options with a String

When using an explicit `gem_binary`, options must be passed as a string. When not using an explicit `gem_binary`, the chef-client is forced to spawn a gems process to install the gems (which uses more system resources) when options are passed as a string. String options are passed verbatim to the gems command and should be specified just as if they were passed on a command line. For example, `--prerelease` for a pre-release gem.

Example

```
gem_package "nokogiri" do
  gem_binary("/opt/ree/bin/gem")
  options("--prerelease --no-format-executable")
end
```

Specify Options with a .gemrc File

Options can be specified in a .gemrc file. By default the `gem_package` resource will use the Ruby interface to install gems which will ignore the .gemrc file. The `gem_package` resource can be forced to use the gems command instead (and to read the .gemrc file) by adding the `gem_binary` attribute to a code block.

Example

```
gem_package "nokogiri" do
  gem_binary "gem"
end
```

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:reconfig`	Use to reconfigure a package. This action requires a response file.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package. (Debian platform only; for other platforms, use the `:remove` action.)

Attributes

This resource has the following attributes:

Attribute	Description
`allow_downgrade`	**yum_package** resource only. Use to downgrade a package to satisfy requested version requirements. Default value: `false`.
`arch`	**yum_package** resource only. The architecture of the package that will be installed or upgraded. (This value can also be passed as part of the package name.)
`flush_cache`	**yum_package** resource only. Yum automatically synchronizes remote metadata to a local cache. The chef-client creates a copy of the local cache, and then stores it in-memory during the chef-client run. The in-memory cache allows packages to be installed during the chef-client run without the need to continue synchronizing the remote metadata to the local cache while the chef-client run is in-progress. Use this attribute to flush the in-memory cache before or after a Yum operation that installs, upgrades, or removes a package. Default value: `{ :before => false, :after => false }`. **Note** The `flush_cache` attribute does not flush the local Yum cache! Use Yum tools—`yum clean headers`, `yum clean packages`, `yum clean all`—to clean the local Yum cache.
`gem_binary`	An attribute for the `gem_package` provider that is used to specify a gems binary.
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for

Attribute	Description
	more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

The following providers are available. Use the short name to call the provider from a recipe:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Apt`	`apt_package`	
`Chef::Provider::Package::Dpkg`	`dpkg_package`	Can be used with the `options` attribute.
`Chef::Provider::Package::EasyInstall`	`easy_install_package`	
`Chef::Provider::Package::Freebsd`	`freebsd_package`	
`Chef::Provider::Package::Ips`	`ips_package`	
`Chef::Provider::Package::Macports`	`macports_package`	
`Chef::Provider::Package::Pacman`	`pacman_package`	
`Chef::Provider::Package::Portage`	`portage_package`	Can be used with the `options` attribute.
`Chef::Provider::Package::Rpm`	`rpm_package`	Can be used with the `options` attribute.
`Chef::Provider::Package::Rubygems`	`gem_package`	Can be used with the `options` attribute.
`Chef::Provider::Package::Rubygems`	`chef_gem`	Can be used with the `options` attribute.
`Chef::Provider::Package::Smartos`	`smartos_package`	
`Chef::Provider::Package::Solaris`	`solaris_package`	
`Chef::Provider::Package::Windows`	`package`	The provider that is used with the Microsoft Windows platform.
`Chef::Provider::Package::Yum`	`yum_package`	
`Chef::Provider::Package::Zypper`	`package`	The provider that is used with the openSUSE platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a gems file for use in recipes

```
chef_gem "right_aws" do
  action :install
end

require 'right_aws'
```

Install a gems file from the local file system

```
gem_package "right_aws" do
  source "/tmp/right_aws-1.11.0.gem"
  action :install
end
```

Install a package

```
package "tar" do
  action :install
end
```

Install a package version

```
package "tar" do
  version "1.16.1-1"
  action :install
end
```

Install a package with options

```
package "debian-archive-keyring" do
  action :install
  options "--force-yes"
end
```

Install a package with a response_file

Use of a `response_file` is only supported on Debian and Ubuntu at this time. Providers need to be written to support the use of a `response_file`, which contains debconf answers to questions normally asked by the package manager on installation. Put the file in `/files/default` of the cookbook where the package is specified and the chef-client will use the **cookbook_file** resource to retrieve it.

To install a package with a `response_file`:

```
package "sun-java6-jdk" do
  response_file "java.seed"
end
```

Install a package using a specific provider

```
package "tar" do
  action :install
  source "/tmp/tar-1.16.1-1.rpm"
  provider Chef::Provider::Package::Rpm
end
```

Install a specified architecture using a named provider

```
yum_package "glibc-devel" do
  arch "i386"
end
```

Purge a package

```
package "tar" do
  action :purge
end
```

Remove a package

```
package "tar" do
  action :remove
end
```

Upgrade a package

```
package "tar" do
  action :upgrade
end
```

Avoid unnecessary string interpolation

Do this:

```
package "mysql-server" do
  version node['mysql']['version']
  action :install
end
```

and not this:

```
package "mysql-server" do
  version "#{node['mysql']['version']}"
  action :install
end
```

Install a package in a platform

The following example shows how to use the **package** resource to install an application named `app` and ensure that the correct packages are installed for the correct platform:

```
package "app_name" do
  action :install
end
```

```
case node[:platform]
when "ubuntu","debian"
  package "app_name-doc" do
    action :install
  end
when "centos"
  package "app_name-html" do
    action :install
  end
end
```

Install sudo, then configure /etc/sudoers/ file

The following example shows how to install sudo and then configure the `/etc/sudoers` file:

```
#  the following code sample comes from the ``default`` recipe in the ``sudo`` cookbook: https://github.com/

package 'sudo' do
  action :install
end

if node['authorization']['sudo']['include_sudoers_d']
  directory '/etc/sudoers.d' do
    mode        '0755'
    owner       'root'
    group       'root'
    action      :create
  end

  cookbook_file '/etc/sudoers.d/README' do
    source      "README"
    mode        '0440'
    owner       'root'
    group       'root'
    action      :create
  end
end

template '/etc/sudoers' do
  source 'sudoers.erb'
  mode '0440'
  owner 'root'
  group platform?('freebsd') ? 'wheel' : 'root'
  variables(
    :sudoers_groups => node['authorization']['sudo']['groups'],
    :sudoers_users => node['authorization']['sudo']['users'],
    :passwordless => node['authorization']['sudo']['passwordless']
  )
end
```

where

- the **package** resource is used to install sudo
- the `if` statement is used to ensure availability of the `/etc/sudoers.d` directory
- the **template** resource tells the chef-client where to find the `sudoers` template
- the `variables` attribute is a hash that passes values to template files (that are located in the `templates/` directory for the cookbook

Use a case statement to specify the platform

The following example shows how to use a case statement to tell the chef-client which platforms and packages to install using cURL.

```
package "curl"
  case node[:platform]
  when "redhat", "centos"
    package "package_1"
    package "package_2"
    package "package_3"
  when "ubuntu", "debian"
    package "package_a"
    package "package_b"
    package "package_c"
  end
end
```

where `node[:platform]` for each node is identified by Ohai during every chef-client run. For example:

```
package "curl"
  case node[:platform]
  when "redhat", "centos"
    package "zlib-devel"
    package "openssl-devel"
    package "libc6-dev"
  when "ubuntu", "debian"
    package "openssl"
    package "pkg-config"
    package "subversion"
  end
end
```

Use symbols to reference attributes

Symbols may be used to reference attributes:

```
package "mysql-server" do
  version node[:mysql][:version]
  action :install
end
```

instead of strings:

```
package "mysql-server" do
  version node['mysql']['version']
  action :install
end
```

Use a whitespace array to simplify a recipe

The following examples show different ways of doing the same thing. The first shows a series of packages that will be upgraded:

```
package "package-a" do
  action :upgrade
end

package "package-b" do
  action :upgrade
end

package "package-c" do
  action :upgrade
end

package "package-d" do
  action :upgrade
end
```

and the next uses a single **package** resource and a whitespace array (`%w`):

```
%w{package-a package-b package-c package-d}.each do |pkg|
  package pkg do
    action :upgrade
  end
end
```

where `|pkg|` is used to define the name of the resource, but also to ensure that each item in the whitespace array has its own name.

pacman_package

Use the **pacman_package** resource to manage packages (using pacman) on the Arch Linux platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **pacman_package** resource in a recipe is as follows:

```
pacman_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `pacman_package` tells the chef-client to use the `Chef::Provider::Package::Pacman` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Pacman`	`pacman_package`	The provider that is used with the Arch Linux platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
pacman_package "name of package" do
  action :install
end
```

perl

Use the **perl** resource to execute scripts using the Perl interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **perl** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **perl** resource in a recipe is as follows:

```
perl "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
```

```
end
```

where

- `perl` tells the chef-client to use the `Chef::Resource::Script::Perl` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Perl`	`perl`	The provider that is used with the Perl command interpreter.

Examples

None.

portage_package

Use the **portage_package** resource to manage packages for the Gentoo platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **portage_package** resource in a recipe is as follows:

```
portage_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `portage_package` tells the chef-client to use the `Chef::Provider::Package::Portage` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Portage`	`portage_package`	The provider that is used with the Gentoo platform. Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
portage_package "name of package" do
  action :install
end
```

perl

A resource defines the desired state for a single configuration item present on a node that is under management by Chef. A resource collection—one (or more) individual resources—defines the desired state for the entire node. During every chef-client run, the current state of each resource is tested, after which the chef-client will take any steps that are necessary to repair the node and bring it back into the desired state.

Use the **perl** resource to execute scripts using the Perl interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **perl** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **perl** resource in a recipe is as follows:

```
perl "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `perl` tells the chef-client to use the `Chef::Resource::Script::Perl` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.

Attribute	Description
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Guards

A guard attribute can be used to evaluate the state of a node during the execution phase of the chef-client run. Based on the results of this evaluation, a guard attribute is then used to tell the chef-client if it should continue executing a resource. A guard attribute accepts either a string value or a Ruby block value:

- A string is executed as a shell command. If the command returns `0`, the guard is applied. If the command returns any other value, then the guard attribute is not applied.
- A block is executed as Ruby code that must return either `true` or `false`. If the block returns `true`, the guard attribute is applied. If the block returns `false`, the guard attribute is not applied.

A guard attribute is useful for ensuring that a resource is idempotent by allowing that resource to test for the desired state as it is being executed, and then if the desired state is present, for the chef-client to do nothing.

Attributes

The following attributes can be used to define a guard that is evaluated during the execution phase of the chef-client run:

Guard	Description
`not_if`	Use to prevent a resource from executing when the condition returns `true`.
`only_if`	Use to allow a resource to execute only if the condition returns `true`.

Arguments

The following arguments can be used with the `not_if` or `only_if` guard attributes:

Argument	Description
`:user`	Use to specify the user that a command will run as. For example: `not_if "grep adam /etc/passwd", :user => 'adam'`
`:group`	Use to specify the group that a command will run as. For example: `not_if "grep adam /etc/passwd", :group => 'adam'`
`:environment`	Use to specify a Hash of environment variables to be set. For example: `not_if "grep adam /etc/passwd", :environment => { 'HOME' => "/home/adam" }`
`:cwd`	Use to set the current working directory before running a command. For example: `not_if "grep adam passwd", :cwd => '/etc'`
`:timeout`	Use to set a timeout for a command. For example: `not_if "sleep 10000", :timeout => 10`

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.

Long name	Short name	Notes
`Chef::Provider::Script::Perl`	`perl`	The provider that is used with the Perl command interpreter.

Examples

None.

portage_package

A resource defines the desired state for a single configuration item present on a node that is under management by Chef. A resource collection—one (or more) individual resources—defines the desired state for the entire node. During every chef-client run, the current state of each resource is tested, after which the chef-client will take any steps that are necessary to repair the node and bring it back into the desired state.

Use the **portage_package** resource to manage packages for the Gentoo platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **portage_package** resource in a recipe is as follows:

```
portage_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `portage_package` tells the chef-client to use the `Chef::Provider::Package::Portage` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.

Attribute	Description
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Portage`	`portage_package`	The provider that is used with the Gentoo platform. Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
portage_package "name of package" do
  action :install
end
```

powershell_script

Use the **powershell_script** resource to execute a script using the Windows PowerShell interpreter, much like how the **script** and **script**-based resources—**bash**, **csh**, **perl**, **python**, and **ruby**—are used. The **powershell_script** is specific to the Microsoft Windows platform and the Windows PowerShell interpreter. This resource creates and executes a temporary file (similar to how the **script** resource behaves), rather than running the command inline. This resource includes actions (`:run` and `:nothing;`) and attributes (`creates`, `cwd`, `environment`, `group`, `path`, `timeout`, and `user`) that are inherited from the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Syntax

The syntax for using the **powershell_script** resource in a recipe is as follows:

```
powershell_script "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `powershell_script` tells the chef-client to use the `Chef::Provider::PowershellScript` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
powershell_script "name_of_script" do
  cwd Chef::Config[:file_cache_path]
  code <<-EOH
     # some script goes here
  EOH
end
```

Actions

This resource has the following actions:

Action	Description
`:run`	Default. Use to run the script.

Attributes

This resource has the following attributes:

Attribute	Description
`architecture`	The architecture of the process under which a script is executed. Possible values: `:x86` (for 32-bit processes) and `:x86_64` (for 64-bit processes). If these values are not provided in a recipe, the chef-client will default to the correct value for the architecture, as determined by Ohai. An exception will be raised when anything other than `:x86` is specified for a 32-bit process.
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)

`convert_boolean_return`	Use to return `0` if the last line of a command is evaluated to be true or to return `1` if the last line is evaluated to be false. Default value: `false`.
	
	When the `guard_intrepreter` common attribute is set to `:powershell_script`, a string command will be evaluated as if this value were set to `true`. This is because the behavior of this attribute is similar to the value of the `"$?"` expression common in UNIX interpreters. For example, this:
	
	`powershell_script 'make_safe_backup' do`
	`  guard_interpreter :powershell_script`
	`  code 'cp ~/data/nodes.json ~/data/nodes.bak'`
	`  not_if 'test-path ~/data/nodes.bak'`
	`end`
	
	is similar to:
	
	`bash 'make_safe_backup' do`
	`  code 'cp ~/data/nodes.json ~/data/nodes.bak'`
	`  not_if 'test -e ~/data/nodes.bak'`
	`end`

`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked. Default value: `[ -NoLogo, -NonInteractive, -NoProfile, -ExecutionPolicy RemoteSigned, -InputFormat None, -File ]`.
`interpreter`	The script interpreter to be used during code execution.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::PowershellScript`	`powershell_script`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Write to an interpolated path

```
powershell_script "write-to-interpolated-path" do
  code <<-EOH
  $stream = [System.IO.StreamWriter] "#{Chef::Config[:file_cache_path]}/powershell-test.txt"
  $stream.WriteLine("In #{Chef::Config[:file_cache_path]}...word.")
  $stream.close()
  EOH
end
```

Change the working directory

```
powershell_script "cwd-then-write" do
  cwd Chef::Config[:file_cache_path]
  code <<-EOH
  $stream = [System.IO.StreamWriter] "C:/powershell-test2.txt"
  $pwd = pwd
  $stream.WriteLine("This is the contents of: $pwd")
  $dirs = dir
  foreach ($dir in $dirs) {
    $stream.WriteLine($dir.fullname)
  }
  $stream.close()
  EOH
end
```

Change the working directory in Microsoft Windows

```
powershell_script "cwd-to-win-env-var" do
  cwd "%TEMP%"
  code <<-EOH
  $stream = [System.IO.StreamWriter] "./temp-write-from-chef.txt"
  $stream.WriteLine("chef on windows rox yo!")
  $stream.close()
  EOH
end
```

Pass an environment variable to a script

```
powershell_script "read-env-var" do
  cwd Chef::Config[:file_cache_path]
  environment ({'foo' => 'BAZ'})
  code <<-EOH
  $stream = [System.IO.StreamWriter] "./test-read-env-var.txt"
  $stream.WriteLine("FOO is $foo")
  $stream.close()
  EOH
end
```

python

Use the **python** resource to execute scripts using the Python interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **python** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **python** resource in a recipe is as follows:

```
python "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `python` tells the chef-client to use the `Chef::Resource::Script::Python` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)

Attribute	Description
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Python`	`python`	The provider that is used with the Python command interpreter.

Examples

None.

registry_key

Use the **registry_key** resource to create and delete registry keys in Microsoft Windows.

64-bit versions of Microsoft Windows have a 32-bit compatibility layer in the registry that reflects and redirects certain keys (and their sub-keys) into specific locations. By default, the registry functionality will default to the machine architecture of the system that is being configured. The chef-client can access any reflected or redirected registry key. The chef-client can write to any 64-bit registry location. (This behavior is not affected by the chef-client running as a 32-bit application.) For more information, see: http://msdn.microsoft.com/en-us/library/windows/desktop/aa384235(v=vs.85).aspx.

Syntax

The syntax for using the **registry_key** resource in a recipe is as follows:

```
registry_key "name" do
  attribute "value" # see attributes section below
  ...
  values [{
    :name => "name",
    :type => :string,
    :data => "data"
    },
    {
    :name => "name",
    :type => :string,
    :data => "data"
    },
    ...
    ]
  action :action # see actions section below
end
```

where

- `registry_key` tells the chef-client to use the `Chef::Provider::Windows::Registry` provider during the chef-client run

- `name` is the name of the resource block; when the `key` attribute is not specified as part of a recipe, `name` is also path to the location in which a registry key is created or from which a registry key is deleted
- `attribute` is zero (or more) of the attributes that are available for this resource
- `values` is a hash that contains at least one registry key to be created or deleted. Each registry key in the hash is grouped by brackets in which the `:name`, `:type`, and `:data` values for that registry key are specified.
- `:type` represents the values available for registry keys in Microsoft Windows. Use `:binary` for REG_BINARY, `:string` for REG_SZ, `:multi_string` for REG_MULTI_SZ, `:expand_string` for REG_EXPAND_SZ, `:dword` for REG_DWORD, `:dword_big_endian` for REG_DWORD_BIG_ENDIAN, or `:qword` for REG_QWORD.

  Warning

  `:multi_string` must be an array, even if there is only a single string.

- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example, a Microsoft Windows registry key named “System” will get a new value called “NewRegistryKeyValue” and a multi-string value named “foo bar”:

```
registry_key "HKEY_LOCAL_MACHINE\\...\\System" do
  values [{
    :name => "NewRegistryKeyValue",
    :type => :multi_string,
    :data => ['foo\0bar\0\0']
  }]
  action :create
end
```

Or, using multiple registry key entries to configure a single resource block with key values based on node attributes:

```
registry_key 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\name_of_registry_key' do
  values [{:name => 'key_name', :type => :string, :data => 'C:\Windows\System32\file_name.bmp'},
          {:name => 'key_name', :type => :string, :data => node['node_name']['attribute']['value']},
          {:name => 'key_name', :type => :string, :data => node['node_name']['attribute']['value']}
         ]
  action :create
end
```

Registry Key Path Separators

A Microsoft Windows registry key can be used as a string in Ruby code, such as when a registry key is used as the name of a recipe. In Ruby, when a registry key is enclosed in a double-quoted string (`" "`), the same backslash character (`\`) that is used to define the registry key path separator is also used in Ruby to define an escape character. Therefore, the registry key path separators must be escaped. For example, the following registry key:

```
HKCU\SOFTWARE\Policies\Microsoft\Windows\CurrentVersion\Themes
```

will not work when it is defined like this:

```
registry_key "HKCU\SOFTWARE\Policies\Microsoft\Windows\CurrentVersion\Themes" do
  ...
  action :some_action
end
```

but will work when the path separators are escaped properly:

```
registry_key "HKCU\\SOFTWARE\\Policies\\Microsoft\\Windows\\CurrentVersion\\Themes" do
  ...
  action :some_action
end
```

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a registry key. If a registry key already exists (but does not match), use to update that registry key to match.
`:create_if_missing`	Use to create a registry key if it does not exist. Also, use to create a registry key value if it does not exist.
`:delete`	Use to delete the specified values for a registry key.
`:delete_key`	Use to delete the specified registry key and all of its subkeys.

Note

Be careful when using the `:delete_key` action with the `recursive` attribute. This will delete the registry key, all of its subkeys and all of the values associated with them. This cannot be undone by the chef-client.

Attributes

This resource has the following attributes:

Attribute	Description
`architecture`	The architecture of the node for which keys will be created or deleted. Possible values: `:i386` (for nodes with a 32-bit registry), `:x86_64` (for nodes with a 64-bit registry), and `:machine` (to have the chef-client determine the architecture during the chef-client run). Default value: `:machine`. In order to read or write 32-bit registry keys on 64-bit machines running Microsoft Windows, the `architecture` attribute must be set to `:i386`. The `:x86_64` value can be used to force writing to a 64-bit registry location, but this value is less useful than the default (`:machine`) because the chef-client will return an exception if `:x86_64` is used and the machine turns out to be a 32-bit machine (whereas with `:machine`, the chef-client will be able to access the registry key on the 32-bit machine). **Note** The `ARCHITECTURE` attribute should only specify `:x86_64` or `:i386` when it is necessary to write 32-bit (`:i386`) or 64-bit (`:x86_64`) values on a 64-bit machine. `ARCHITECTURE` will default to `:machine` unless a specific value is given.
`key`	The path to the location in which a registry key will be created or from which a registry key will be deleted. Default value: the `name` of the resource block. (See “Syntax” section above for more information.) The path must include the registry hive, which can be specified either as its full name or as the 3- or 4-letter abbreviation. For example, both `HKLM\SECURITY` and `HKEY_LOCAL_MACHINE\SECURITY` are both valid and equivalent. The following hives are valid: `HKEY_LOCAL_MACHINE`, `HKLM`, `HKEY_CURRENT_CONFIG`, `HKCC`, `HKEY_CLASSES_ROOT`, `HKCR`, `HKEY_USERS`, `HKU`, `HKEY_CURRENT_USER`, and `HKCU`.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`recursive`	When creating a key, this value specifies that the required keys for the specified path will be created. When using the `:delete_key` action in a recipe, and if the registry key has subkeys, then the value for this attribute should be set to `true`. **Note** Be careful when using the `:delete_key` action with the `recursive` attribute. This will delete the registry key, all of its subkeys and all of the values associated with them. This cannot be undone by the chef-client.
`values`	An array of hashes, where each Hash contains the values that will be set under a registry key. Each Hash must contain `:name`, `:type`, and `:data` (and must contain no other key values). `:type` represents the values available for registry keys in Microsoft Windows. Use `:binary` for REG_BINARY, `:string` for REG_SZ, `:multi_string` for REG_MULTI_SZ, `:expand_string` for REG_EXPAND_SZ, `:dword` for REG_DWORD, `:dword_big_endian` for REG_DWORD_BIG_ENDIAN, or `:qword` for REG_QWORD. **Warning** `:multi_string` must be an array, even if there is only a single string.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Windows::Registry`	`registry_key`	The default provider for the Microsoft Windows platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create a registry key

```
registry_key "HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\System" do
  values [{
    :name => "EnableLUA",
    :type => :dword,
    :data => 0
  }]
  action :create
end
```

Delete a registry key value

```
registry_key "HKEY_LOCAL_MACHINE\\SOFTWARE\\Policies\\Microsoft\\Windows\\WindowsUpdate\\AU" do
  values [{
    :name => "NoAutoRebootWithLoggedOnUsers",
    :type => :dword
    }]
  action :delete
end
```

Delete a registry key and its subkeys, recursively

```
registry_key "HKCU\\SOFTWARE\\Policies\\Microsoft\\Windows\\CurrentVersion\\Themes" do
  recursive true
  action :delete_key
end
```

Note

Be careful when using the `:delete_key` action with the `recursive` attribute. This will delete the registry key, all of its subkeys and all of the values associated with them. This cannot be undone by the chef-client.

Use re-directed keys

In 64-bit versions of Microsoft Windows, `HKEY_LOCAL_MACHINE\SOFTWARE\Example` is a re-directed key. In the following examples, because `HKEY_LOCAL_MACHINE\SOFTWARE\Example` is a 32-bit key, the output will be "Found 32-bit key" if they are run on a version of Microsoft Windows that is 64-bit:

```
registry_key "HKEY_LOCAL_MACHINE\\SOFTWARE\\Example" do
  architecture :i386
  recursive true
  action :create
end
```

or:

```
registry_key "HKEY_LOCAL_MACHINE\\SOFTWARE\\Example" do
  architecture :x86_64
  recursive true
  action :delete_key
end
```

or:

```
ruby_block "check 32-bit" do
  block do
    puts "Found 32-bit key"
  end
  only_if { registry_key_exists?("HKEY_LOCAL_MACHINE\SOFTWARE\\Example", :i386) }
end
```

or:

```
ruby_block "check 64-bit" do
  block do
    puts "Found 64-bit key"
  end
  only_if { registry_key_exists?("HKEY_LOCAL_MACHINE\\SOFTWARE\\Example", :x86_64) }
end
```

Set proxy settings to be the same as those used by the chef-client

```
proxy = URI.parse(Chef::Config[:http_proxy])
registry_key 'HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings' do
  values [{:name => 'ProxyEnable', :type => :reg_dword, :data => 1},
          {:name => 'ProxyServer', :data => "#{proxy.host}:#{proxy.port}"},
          {:name => 'ProxyOverride', :type => :reg_string, :data => <local>},
         ]
  action :create
end
```

remote_directory

Use the **remote_directory** resource to incrementally transfer a directory from a cookbook to a node. The directory that is copied from the cookbook should be located under `COOKBOOK_NAME/files/default/REMOTE_DIRECTORY`. The **remote_directory** resource will obey file specificity.

Syntax

The syntax for using the **remote_directory** resource in a recipe is as follows:

```
remote_directory "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `remote_directory` tells the chef-client to use the `Chef::Provider::Directory::RemoteDirectory` provider during the chef-client run
- `name` is the path to the location below which the chef-client will manage directories
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a directory and/or the contents of that directory. If a directory or its contents already exist (but does not match), use to update that directory or its contents to match.
`:create_if_missing`	Use to create a directory and/or the contents of that directory, but only if it does not exist.
`:delete`	Use to delete a directory, including the contents of that directory.

Attributes

This resource has the following attributes:

Attribute	Description
`cookbook`	The cookbook in which a file is located (if it is not located in the current cookbook). The default value is the current cookbook.
`files_backup`	The number of backup copies to keep for files in the directory. Default value: `5`.
`files_group`	Use to configure group permissions for files. A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`files_mode`	The octal mode for a file. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`files_owner`	Use to configure owner permissions for files. A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`group`	Use to configure permissions for directories. A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is

Attribute	Description
	not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`mode`	A quoted string that defines the octal mode for a directory. If `mode` is not specified and if the directory already exists, the existing mode on the directory is used. If `mode` is not specified, the directory does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the directory will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`overwrite`	Use to overwrite a file when it is different. Default value: `true`.
`owner`	Use to configure permissions for directories. A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`path`	The path to the directory. Using a fully qualified path is recommended, but is not always required. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`purge`	Use to purge extra files found in the target directory. Default value: `false`.
`recursive`	Use to create or delete directories recursively. Default value: `true`; the chef-client must be able to create the directory structure, including parent directories (if missing), as defined in `COOKBOOK_NAME/files/default/REMOTE_DIRECTORY`.
`rights`	Microsoft Windows only. The permissions for users and groups in a Microsoft Windows environment. For example: `rights <permissions>, <principal>, <options>` where `<permissions>` specifies the rights granted to the principal, `<principal>` is the group or user name, and `<options>` is a Hash with one (or more) advanced rights options.
`source`	The base name of the source file (and inferred from the `path` attribute).

Recursive Directories

The **remote_directory** resource can be used to recursively create the path outside of remote directory structures, but the permissions of those outside paths are not managed. This is because the `recursive` attribute only applies `group`, `mode`, and `owner` attribute values to the remote directory itself and any inner directories the resource copies.

A directory structure:

```
/foo
  /bar
    /baz
```

The following example shows a way create a file in the `/baz` directory:

```
remote_directory "/foo/bar/baz" do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

But with this example, the `group`, `mode`, and `owner` attribute values will only be applied to `/baz`. Which is fine, if that's what you want. But most of the time, when the entire `/foo/bar/baz` directory structure is not there, you must be explicit about each directory. For example:

```
%w[ /foo /foo/bar /foo/bar/baz ].each do |path|
  remote_directory path do
    owner 'root'
    group 'root'
    mode '0755'
  end
end
```

This approach will create the correct hierarchy—`/foo`, then `/bar` in `/foo`, and then `/baz` in `/bar`—and also with the correct attribute values for `group`, `mode`, and `owner`.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Directory::RemoteDirectory`	`remote_directory`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Recursively transfer a directory from a remote location

```
# create up to 10 backups of the files, set the files owner different from the directory.
remote_directory "/tmp/remote_something" do
  source "something"
  files_backup 10
  files_owner 'root'
  files_group 'root'
  files_mode '0644'
  owner 'nobody'
  group 'nobody'
  mode '0755'
end
```

Use with the chef_handler lightweight resource

The following example shows how to use the **remote_directory** resource and the **chef_handler** lightweight resource to reboot a handler named `WindowsRebootHandler`:

```
#  the following code sample comes from the ``reboot_handler`` recipe in the ``windows`` cookbook: https://g

remote_directory node['chef_handler']['handler_path'] do
  source 'handlers'
  recursive true
  action :create
end

chef_handler 'WindowsRebootHandler' do
  source "#{node['chef_handler']['handler_path']}/windows_reboot_handler.rb"
  arguments node['windows']['allow_pending_reboots']
  supports :report => true, :exception => false
  action :enable
end
```

remote_file

Use the **remote_file** resource to transfer a file from a remote location using file specificity. This resource is similar to the **file** resource.

Note

Fetching files from the `files/` directory in a cookbook should be done with the **cookbook_file** resource.

Syntax

The syntax for using the **remote_file** resource in a recipe is as follows:

```
remote_file "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `remote_file` tells the chef-client to use the `Chef::Provider::File::RemoteFile` provider during the chef-client run
- `name` is the name of the resource block; when the `path` attribute is not specified as part of a recipe, `name` is also the path to the remote file
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
remote_file "#{Chef::Config[:file_cache_path]}/large-file.tar.gz" do
  source "http://www.example.org/large-file.tar.gz"
end
```

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a file. If a file already exists (but does not match), use to update that file to match.
`:create_if_missing`	Use to create a file only if the file does not exist. (When the file exists, nothing happens.)
`:delete`	Use to delete a file.
`:touch`	Use to touch a file. This updates the access (atime) and file modification (mtime) times for a file. (This action may be used with this resource, but is typically only used with the **file** resource.)

Attributes

This resource has the following attributes:

Attribute	Description
`atomic_update`	Use to perform atomic file updates on a per-resource basis. Set to `true` for atomic file updates. Set to `false` for non-atomic file updates. (This setting overrides `file_atomic_update`, which is a global setting found in the client.rb file.) Default value: `true`.
`backup`	The number of backups to be kept. Set to `false` to prevent backups from being kept. Default value: `5`.
`checksum`	Optional, see `use_conditional_get`. The SHA-256 checksum of the file. Use to prevent the **remote_file** resource from re-downloading a file. When the local file matches the checksum, the chef-client will not download it.
`force_unlink`	Use to specify how the chef-client handles certain situations when the target file turns out not to be a file. For example, when a target file is actually a symlink. Set to `true` to have the chef-client delete the non-file target and replace it with the specified file. Set to `false` for the chef-client to raise an error. Default value: `false`.
`ftp_active_mode`	Use to specify if the chef-client will use active or passive FTP. Set to `true` to use active FTP. Default value: `false`.
`group`	A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`headers`	A Hash of custom headers. Default value: `{}`.
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`manage_symlink_source`	Use to have the chef-client detect and manage the source file for a symlink. Possible values: `nil`, `true`, or `false`. When this value is set to `nil`, the chef-client will manage a symlink's source file and emit a warning. When this value is set to `true`, the chef-client will manage a symlink's source file and not emit a warning. Default value: `nil`. The default value will be changed to `false` in a future version.
`mode`	A quoted string that defines the octal mode for a file. If `mode` is not specified and if the file already exists, the existing mode on the file is used. If `mode` is not specified, the file does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the file will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4`

Attribute	Description
	equals `GENERIC_READ`, 2 equals `GENERIC_WRITE`, and 1 equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.
`owner`	A string or ID that identifies the group owner by user name, including fully qualified user names such as `domain\user` or `user@domain`. If this value is not specified, existing owners will remain unchanged and new owner assignments will use the current user (when necessary).
`path`	The path to the file. Using a fully qualified path is recommended, but is not always required. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`rights`	Microsoft Windows only. The permissions for users and groups in a Microsoft Windows environment. For example: `rights <permissions>, <principal>, <options>` where `<permissions>` specifies the rights granted to the principal, `<principal>` is the group or user name, and `<options>` is a Hash with one (or more) advanced rights options.

`source`	Required. The location (URI) of the source file. This value may also specify HTTP (`http://`), FTP (`ftp://`), or local (`file://`) source file locations.
	There are many ways to define the location of a source file. By using a path:
	`source "http://couchdb.apache.org/img/sketch.png"`
	By using a node attribute:
	`source node['nginx']['foo123']['url']`
	By using attributes to define paths:
	`source "#{node['python']['url']}/#{version}/Python-#{version}.tar.bz2"`
	By defining multiple paths for multiple locations:
	`source "http://seapower/spring.png", "http://seapower/has.png", "http://seapower/sprung.png"`
	By defining those same multiple paths as an array:
	`source ["http://seapower/spring.png", "http://seapower/has.png", "http://seapower/sprung.png"]`
	When multiple paths are specified, the chef-client will attempt to download the files in the order listed, stopping after the first successful download.

`use_conditional_get`	Use to enable conditional HTTP requests by using a conditional `GET` (with the If-Modified-Since header) or an opaque identifier (ETag). To use If-Modified-Since headers, `use_last_modified` must also be set to `true`. To use ETag headers, `use_etag` must also be set to `true`. Default value: `true`.
`use_etag`	Use to enable ETag headers. Set to `false` to disable ETag headers. To use this setting, `use_conditional_get` must also be set to `true`. Default value: `true`.
`use_last_modified`	Use to enable If-Modified-Since headers. Set to `false` to disable If-Modified-Since headers. To use this setting, `use_conditional_get` must also be set to `true`. Default value: `true`.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::File::RemoteFile`	`remote_file`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Transfer a file from a URL

```
remote_file "/tmp/testfile" do
  source "http://www.example.com/tempfiles/testfile"
  mode '0644'
  checksum "3a7dac00b1" # A SHA256 (or portion thereof) of the file.
```

```
end
```

Transfer a file only when the source has changed

```
remote_file "/tmp/couch.png" do
  source "http://couchdb.apache.org/img/sketch.png"
  action :nothing
end

http_request "HEAD http://couchdb.apache.org/img/sketch.png" do
  message ""
  url "http://couchdb.apache.org/img/sketch.png"
  action :head
  if File.exists?("/tmp/couch.png")
    headers "If-Modified-Since" => File.mtime("/tmp/couch.png").httpdate
  end
  notifies :create, "remote_file[/tmp/couch.png]", :immediately
end
```

Install a file from a remote location using bash

The following is an example of how to install the `foo123` module for Nginx. This module adds shell-style functionality to an Nginx configuration file and does the following:

- Declares three variables
- Gets the Nginx file from a remote location
- Installs the file using Bash to the path specified by the `src_filepath` variable

```
#  the following code sample is similar to the ``upload_progress_module`` recipe in the ``nginx`` cookbook:

src_filename = "foo123-nginx-module-v#{node['nginx']['foo123']['version']}.tar.gz"
src_filepath = "#{Chef::Config['file_cache_path']}/#{src_filename}"
extract_path = "#{Chef::Config['file_cache_path']}/nginx_foo123_module/#{node['nginx']['foo123']['checksum']

remote_file src_filepath do
  source node['nginx']['foo123']['url']
  checksum node['nginx']['foo123']['checksum']
  owner 'root'
  group 'root'
  mode '0644'
end

bash 'extract_module' do
  cwd ::File.dirname(src_filepath)
  code <<-EOH
    mkdir -p #{extract_path}
    tar xzf #{src_filename} -C #{extract_path}
    mv #{extract_path}/*/* #{extract_path}/
    EOH
  not_if { ::File.exists?(extract_path) }
end
```

Store certain settings

The following recipe shows how an attributes file can be used to store certain settings. An attributes file is located in the `attributes/` directory in the same cookbook as the recipe which calls the attributes file. In this example, the attributes file specifies certain settings for Python that are then used across all nodes against which this recipe will run.

Python packages have versions, installation directories, URLs, and checksum files. An attributes file that exists to support this type of recipe would include settings like the following:

```
default['python']['version'] = '2.7.1'

if python['install_method'] == 'package'
  default['python']['prefix_dir'] = '/usr'
else
  default['python']['prefix_dir'] = '/usr/local'
end

default['python']['url'] = 'http://www.python.org/ftp/python'
default['python']['checksum'] = '80e387...85fd61'
```

and then the methods in the recipe may refer to these values. A recipe that is used to install Python will need to do the following:

- Identify each package to be installed (implied in this example, not shown)
- Define variables for the package `version` and the `install_path`
- Get the package from a remote location, but only if the package does not already exist on the target system
- Use the **bash** resource to install the package on the node, but only when the package is not already installed

```
#  the following code sample comes from the ``oc-nginx`` cookbook on |github|: https://github.com/cookbooks/

version = node['python']['version']
install_path = "#{node['python']['prefix_dir']}/lib/python#{version.split(/(^\d+\.\d+)/)[1]}"
```

```
remote_file "#{Chef::Config[:file_cache_path]}/Python-#{version}.tar.bz2" do
  source "#{node['python']['url']}/#{version}/Python-#{version}.tar.bz2"
  checksum node['python']['checksum']
  mode '0644'
  not_if { ::File.exists?(install_path) }
end

bash "build-and-install-python" do
  cwd Chef::Config[:file_cache_path]
  code <<-EOF
    tar -jxvf Python-#{version}.tar.bz2
    (cd Python-#{version} && ./configure #{configure_options})
    (cd Python-#{version} && make && make install)
  EOF
  not_if { ::File.exists?(install_path) }
end
```

Use the platform_family? method

The following is an example of using the `platform_family?` method in the Recipe DSL to create a variable that can be used with other resources in the same recipe. In this example, `platform_family?` is being used to ensure that a specific binary is used for a specific platform before using the **remote_file** resource to download a file from a remote location, and then using the **execute** resource to install that file by running a command.

```
if platform_family?("rhel")
  pip_binary = "/usr/bin/pip"
else
  pip_binary = "/usr/local/bin/pip"
end

remote_file "#{Chef::Config[:file_cache_path]}/distribute_setup.py" do
  source "http://python-distribute.org/distribute_setup.py"
  mode '0644'
  not_if { ::File.exists?(pip_binary) }
end

execute "install-pip" do
  cwd Chef::Config[:file_cache_path]
  command <<-EOF
    # command for installing Python goes here
    EOF
  not_if { ::File.exists?(pip_binary) }
end
```

where a command for installing Python might look something like:

```
#{node['python']['binary']} distribute_setup.py
#{::File.dirname(pip_binary)}/easy_install pip
```

Specify local Windows file path as a valid URI

When specifying a local Microsoft Windows file path as a valid file URI, an additional forward slash (`/`) is required. For example:

```
remote_file "file:///c:/path/to/file" do
  ...       # other attributes
end
```

route

Use the **route** resource to manage the system routing table in a Linux environment.

Syntax

The syntax for using the **route** resource in a recipe is as follows:

```
route "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `route` tells the chef-client to use the `Chef::Provider::Route` provider during the chef-client run
- `name` is the name of the resource block; when the `target` attribute is not specified as part of a recipe, `name` is also the IP address of the target route
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:add`	Default. Use to add a route.
`:delete`	Use to delete a route.

Attributes

This resource has the following attributes:

Attribute	Description
`device`	The network interface to which the route applies.
`gateway`	The gateway for the route.
`netmask`	The decimal representation of the network mask. For example: `255.255.255.0`.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`target`	The IP address of the target route. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Route`	`route`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Add a host route

```
route "10.0.1.10/32" do
  gateway "10.0.0.20"
  device "eth1"
end
```

Delete a network route

```
route "10.1.1.0/24" do
  gateway "10.0.0.20"
  action :delete
end
```

rpm_package

Use the **rpm_package** resource to manage packages for the RPM Package Manager platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **rpm_package** resource in a recipe is as follows:

```
rpm_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `rpm_package` tells the chef-client to use the `Chef::Provider::Package::Rpm` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Rpm`	`rpm_package`	The provider that is used with the RPM Package Manager platform. Can be used with the `options` attribute.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
rpm_package "name of package" do
  action :install
end
```

ruby

Use the **ruby** resource to execute scripts using the Ruby interpreter. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **ruby** script resource (which is based on the **script** resource) is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **ruby** resource in a recipe is as follows:

```
ruby "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `ruby` tells the chef-client to use the `Chef::Resource::Script::Ruby` provider during the chef-client run
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Ruby`	`ruby`	The provider that is used with the Ruby command interpreter.

Examples

None.

ruby_block

Use the **ruby_block** resource to execute Ruby code during a chef-client run. Ruby code in the `ruby_block` resource is evaluated with other resources during convergence, whereas Ruby code outside of a `ruby_block` resource is evaluated before other resources, as the recipe is compiled.

Syntax

The syntax for using the **ruby_block** resource in a recipe is as follows:

```
ruby_block "name" do
  block do
    # some Ruby code
  end
  action :action # see actions section below
end
```

where

- `ruby_block` tells the chef-client to use the `Chef::Provider::RubyBlock` provider during the chef-client run
- `name` is the name of the resource block; when the `block_name` attribute is not specified as part of a recipe, `name` is also the name of the Ruby block
- `block` is the attribute that is used to define the Ruby block
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:run`	Default. Use to run a Ruby block.
`:create`	The same as `:run`.

Attributes

This resource has the following attributes:

Attribute	Description
`block`	A block of Ruby code.
`block_name`	The name of the Ruby block. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::RubyBlock`	`ruby_block`	The default provider for all platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Re-read configuration data

```
ruby_block "reload_client_config" do
  block do
    Chef::Config.from_file("/etc/chef/client.rb")
  end
  action :run
end
```

Install repositories from a file, trigger a command, and force the internal cache to reload

The following example shows how to install new Yum repositories from a file, where the installation of the repository triggers a creation of the

Yum cache that forces the internal cache for the chef-client to reload:

```
execute "create-yum-cache" do
 command "yum -q makecache"
 action :nothing
end

ruby_block "reload-internal-yum-cache" do
  block do
    Chef::Provider::Package::Yum::YumCache.instance.reload
  end
  action :nothing
end

cookbook_file "/etc/yum.repos.d/custom.repo" do
  source "custom"
  mode '0644'
  notifies :run, "execute[create-yum-cache]", :immediately
  notifies :create, "ruby_block[reload-internal-yum-cache]", :immediately
end
```

Use an if statement with the platform recipe DSL method

The following example shows how an if statement can be used with the `platform?` method in the Recipe DSL to run code specific to Microsoft Windows. The code is defined using the **ruby_block** resource:

```
#  the following code sample comes from the ``client`` recipe in the following cookbook: https://github.com/

if platform?("windows")
  ruby_block "copy libmysql.dll into ruby path" do
    block do
      require 'fileutils'
      FileUtils.cp "#{node['mysql']['client']['lib_dir']}\\libmysql.dll",
        node['mysql']['client']['ruby_dir']
    end
    not_if { File.exist?("#{node['mysql']['client']['ruby_dir']}\\libmysql.dll") }
  end
end
```

Stash a file in a data bag

The following example shows how to use the **ruby_block** resource to stash a BitTorrent file in a data bag so that it can be distributed to nodes in the organization.

```
#  the following code sample comes from the ``seed`` recipe in the following cookbook: https://github.com/ma

ruby_block "share the torrent file" do
  block do
    f = File.open(node['bittorrent']['torrent'],'rb')
    #read the .torrent file and base64 encode it
    enc = Base64.encode64(f.read)
    data = {
      'id'=>bittorrent_item_id(node['bittorrent']['file']),
      'seed'=>node.ipaddress,
      'torrent'=>enc
    }
    item = Chef::DataBagItem.new
    item.data_bag('bittorrent')
    item.raw_data = data
    item.save
  end
  action :nothing
  subscribes :create, "bittorrent_torrent[#{node['bittorrent']['torrent']}]", :immediately
end
```

Update the /etc/hosts file

The following example shows how the **ruby_block** resource can be used to update the `/etc/hosts` file:

```
#  the following code sample comes from the ``ec2`` recipe in the following cookbook: https://github.com/ops

ruby_block "edit etc hosts" do
  block do
    rc = Chef::Util::FileEdit.new("/etc/hosts")
    rc.search_file_replace_line(/^127\.0\.0\.1 localhost$/,
       "127.0.0.1 #{new_fqdn} #{new_hostname} localhost")
    rc.write_file
  end
end
```

Set environment variables

The following example shows how to use variables within a Ruby block to set environment variables using rbenv.

```
node.set[:rbenv][:root] = rbenv_root
node.set[:ruby_build][:bin_path] = rbenv_binary_path

ruby_block "initialize" do
  block do
    ENV['RBENV_ROOT'] = node[:rbenv][:root]
    ENV['PATH'] = "#{node[:rbenv][:root]}/bin:#{node[:ruby_build][:bin_path]}:#{ENV['PATH']}"
  end
end
```

Set JAVA_HOME

The following example shows how to use a variable within a Ruby block to set the `java_home` environment variable:

```
ruby_block "set-env-java-home" do
  block do
    ENV["JAVA_HOME"] = java_home
  end
end
```

Run specific blocks of Ruby code on specific platforms

The following example shows how the `platform?` method and an if statement can be used in a recipe along with the `ruby_block` resource to run certain blocks of Ruby code on certain platforms:

```
if platform?("ubuntu", "debian", "redhat", "centos", "fedora", "scientific", "amazon")
  ruby_block "update-java-alternatives" do
    block do
      if platform?("ubuntu", "debian") and version == 6
        run_context = Chef::RunContext.new(node, {})
        r = Chef::Resource::Execute.new("update-java-alternatives", run_context)
        r.command "update-java-alternatives -s java-6-openjdk"
        r.returns [0,2]
        r.run_action(:create)
      else

        require "fileutils"
        arch = node['kernel']['machine'] =~ /x86_64/ ? "x86_64" : "i386"
        Chef::Log.debug("glob is #{java_home_parent}/java*#{version}*openjdk*")
        jdk_home = Dir.glob("#{java_home_parent}/java*#{version}*openjdk{,[-\.]#{arch}}")[0]
        Chef::Log.debug("jdk_home is #{jdk_home}")

        if File.exists? java_home
          FileUtils.rm_f java_home
        end
        FileUtils.ln_sf jdk_home, java_home

        cmd = Chef::ShellOut.new(
              %Q[ update-alternatives --install /usr/bin/java java #{java_home}/bin/java 1;
              update-alternatives --set java #{java_home}/bin/java ]
              ).run_command
           unless cmd.exitstatus == 0 or cmd.exitstatus == 2
          Chef::Application.fatal!("Failed to update-alternatives for openjdk!")
        end
      end
    end
    action :nothing
  end
end
```

Reload the configuration

The following example shows how to reload the configuration of a chef-client using the **remote_file** resource to:

- using an if statement to check whether the plugins on a node are the latest versions
- identify the location from which Ohai plugins are stored
- using the `notifies` attribute and a **ruby_block** resource to trigger an update (if required) and to then reload the client.rb file.

```
directory node[:ohai][:plugin_path] do
  owner 'chef'
  recursive true
end

ruby_block "reload_config" do
  block do
    Chef::Config.from_file("/etc/chef/client.rb")
  end
  action :nothing
end

if node[:ohai].key?(:plugins)
  node[:ohai][:plugins].each do |plugin|
    remote_file node[:ohai][:plugin_path] +"/#{plugin}" do
      source plugin
      owner 'chef'
              notifies :run, "ruby_block[reload_config]", :immediately
    end
  end
end
```

script

Use the **script** resource to execute scripts using a specified interpreter, such as Bash, csh, Perl, Python, or Ruby. This resource may also use any of the actions and attributes that are available to the **execute** resource. Commands that are executed with this resource are (by their nature) not idempotent, as they are typically unique to the environment in which they are run. Use `not_if` and `only_if` to guard this resource for idempotence.

Note

The **script** resource is different from the **ruby_block** resource because Ruby code that is run with this resource is created as a temporary file and executed like other script resources, rather than run inline.

Syntax

The syntax for using the **script** resource in a recipe is as follows:

```
script "name" do
  some_attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `script` tells the chef-client to use one of the following providers during the chef-client run: `Chef::Resource::Script`, `Chef::Resource::Script::Bash`, `Chef::Resource::Script::Csh`, `Chef::Resource::Script::Perl`, `Chef::Resource::Script::Python`, or `Chef::Resource::Script::Ruby`. The provider that is used by the chef-client depends on the platform of the machine on which the run is taking place
- `name` is the name of the resource block; when the `command` attribute is not specified as part of a recipe, `name` is also the name of the command to be executed
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:nothing`	Use to prevent a command from running. This action is used to specify that a command is run only when another resource notifies it.
`:run`	Default. Use to run a script.

Attributes

This resource has the following attributes:

Attribute	Description
`code`	A quoted (" ") string of code to be executed.
`command`	The name of the command to be executed. Default value: the `name` of the resource block (see Syntax section above).
`creates`	Use to prevent a command from creating a file when that file already exists.
`cwd`	The current working directory.
`environment`	A Hash of environment variables in the form of `{"ENV_VARIABLE" => "VALUE"}`. (These variables must exist for a command to be run successfully.)
`flags`	One (or more) command line flags that are passed to the interpreter when a command is invoked.
`group`	The group name or group ID that must be changed before running a command.
`interpreter`	The script interpreter to be used during code execution.
`path`	An array of paths to use when searching for a command. These paths are not added to the command's environment $PATH. The default value uses the system path.
`provider`	Optional. Use to explicitly specify a provider.

Attribute	Description
`returns`	The return value for a command. This may be an array of accepted values. An exception is raised when the return value(s) do not match. Default value: `0`.
`timeout`	The amount of time (in seconds) a command will wait before timing out. Default value: `3600`.
`user`	The user name or user ID that should be changed before running a command.
`umask`	The file mode creation mask, or umask.

Providers

The following providers are available. Use the short name to use the provider in a recipe:

Long name	Short name	Notes
`Chef::Provider::Script`	`script`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Script::Bash`	`bash`	The provider that is used with the Bash command interpreter.
`Chef::Provider::Script::Csh`	`csh`	The provider that is used with the csh command interpreter.
`Chef::Provider::Script::Perl`	`perl`	The provider that is used with the Perl command interpreter.
`Chef::Provider::Script::Python`	`python`	The provider that is used with the Python command interpreter.
`Chef::Provider::Script::Ruby`	`ruby`	The provider that is used with the Ruby command interpreter.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Use a named provider to run a script

```
bash "install_something" do
  user "root"
  cwd "/tmp"
  code <<-EOH
  wget http://www.example.com/tarball.tar.gz
  tar -zxf tarball.tar.gz
  cd tarball
  ./configure
  make
  make install
  EOH
end
```

Run a script

```
script "install_something" do
  interpreter "bash"
  user "root"
  cwd "/tmp"
  code <<-EOH
  wget http://www.example.com/tarball.tar.gz
  tar -zxf tarball.tar.gz
  cd tarball
  ./configure
  make
  make install
  EOH
end
```

or something like:

```
bash "openvpn-server-key" do
  environment("KEY_CN" => "server")
  code <<-EOF
    openssl req -batch -days #{node["openvpn"]["key"]["expire"]} \
      -nodes -new -newkey rsa:#{key_size} -keyout #{key_dir}/server.key \
      -out #{key_dir}/server.csr -extensions server \
      -config #{key_dir}/openssl.cnf
  EOF
  not_if { ::File.exists?("#{key_dir}/server.crt") }
end
```

where `code` contains the OpenSSL command to be run. The `not_if` method tells the chef-client not to run the command if the file already exists.

Install a file from a remote location using bash

The following is an example of how to install the `foo123` module for Nginx. This module adds shell-style functionality to an Nginx configuration file and does the following:

- Declares three variables
- Gets the Nginx file from a remote location
- Installs the file using Bash to the path specified by the `src_filepath` variable

```
#  the following code sample is similar to the ``upload_progress_module`` recipe in the ``nginx`` cookbook:

src_filename = "foo123-nginx-module-v#{node['nginx']['foo123']['version']}.tar.gz"
src_filepath = "#{Chef::Config['file_cache_path']}/#{src_filename}"
extract_path = "#{Chef::Config['file_cache_path']}/nginx_foo123_module/#{node['nginx']['foo123']['checksum']

remote_file src_filepath do
  source node['nginx']['foo123']['url']
  checksum node['nginx']['foo123']['checksum']
  owner 'root'
  group 'root'
  mode '0644'
end

bash 'extract_module' do
  cwd ::File.dirname(src_filepath)
  code <<-EOH
    mkdir -p #{extract_path}
    tar xzf #{src_filename} -C #{extract_path}
    mv #{extract_path}/*/* #{extract_path}/
    EOH
  not_if { ::File.exists?(extract_path) }
end
```

Install an application from git using bash

The following example shows how Bash can be used to install a plug-in for rbenv named `ruby-build`, which is located in git version source control. First, the application is synchronized, and then Bash changes its working directory to the location in which `ruby-build` is located, and then runs a command.

```
 git "#{Chef::Config[:file_cache_path]}/ruby-build" do
   repository "git://github.com/sstephenson/ruby-build.git"
   reference "master"
   action :sync
 end

 bash "install_ruby_build" do
   cwd "#{Chef::Config[:file_cache_path]}/ruby-build"
   user "rbenv"
   group "rbenv"
   code <<-EOH
     ./install.sh
     EOH
   environment 'PREFIX' => "/usr/local"
end
```

To read more about `ruby-build`, see here: https://github.com/sstephenson/ruby-build.

Store certain settings

The following recipe shows how an attributes file can be used to store certain settings. An attributes file is located in the `attributes/` directory in the same cookbook as the recipe which calls the attributes file. In this example, the attributes file specifies certain settings for Python that are then used across all nodes against which this recipe will run.

Python packages have versions, installation directories, URLs, and checksum files. An attributes file that exists to support this type of recipe would include settings like the following:

```
default['python']['version'] = '2.7.1'

if python['install_method'] == 'package'
  default['python']['prefix_dir'] = '/usr'
else
  default['python']['prefix_dir'] = '/usr/local'
end

default['python']['url'] = 'http://www.python.org/ftp/python'
default['python']['checksum'] = '80e387...85fd61'
```

and then the methods in the recipe may refer to these values. A recipe that is used to install Python will need to do the following:

- Identify each package to be installed (implied in this example, not shown)
- Define variables for the package `version` and the `install_path`

- Get the package from a remote location, but only if the package does not already exist on the target system
- Use the **bash** resource to install the package on the node, but only when the package is not already installed

```
#  the following code sample comes from the ``oc-nginx`` cookbook on |github|: https://github.com/cookbooks/

version = node['python']['version']
install_path = "#{node['python']['prefix_dir']}/lib/python#{version.split(/(^\d+\.\d+)/)[1]}"

remote_file "#{Chef::Config[:file_cache_path]}/Python-#{version}.tar.bz2" do
  source "#{node['python']['url']}/#{version}/Python-#{version}.tar.bz2"
  checksum node['python']['checksum']
  mode '0644'
  not_if { ::File.exists?(install_path) }
end

bash "build-and-install-python" do
  cwd Chef::Config[:file_cache_path]
  code <<-EOF
    tar -jxvf Python-#{version}.tar.bz2
    (cd Python-#{version} && ./configure #{configure_options})
    (cd Python-#{version} && make && make install)
  EOF
  not_if { ::File.exists?(install_path) }
end
```

service

Use the **service** resource to manage a service.

Syntax

The syntax for using the **service** resource in a recipe is as follows:

```
service "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `service` tells the chef-client to use one of the following providers during the chef-client run: `Chef::Provider::Service::Init`, `Chef::Provider::Service::Init::Debian`, `Chef::Provider::Service::Upstart`, `Chef::Provider::Service::Init::Freebsd`, `Chef::Provider::Service::Init::Gentoo`, `Chef::Provider::Service::Init::Redhat`, `Chef::Provider::Service::Solaris`, `Chef::Provider::Service::Windows`, or `Chef::Provider::Service::Macosx`. The chef-client will detect the platform at the start of the run based on data collected by Ohai. After the platform is identified, the chef-client will determine the correct provider
- `name` is the name of the resource block; when the `service_name` attribute is not specified as part of a recipe, `name` is also the name of the service
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

Attributes

This resource has the following attributes:

Attribute	Description
`init_command`	The path to the init script associated with the service. This is typically `/etc/init.d/SERVICE_NAME`. The `init_command` attribute can be used to prevent the need to specify overrides for the `start_command`, `stop_command`, and `restart_command` attributes. Default value: `nil`.
`pattern`	The pattern to look for in the process table. Default value: `service_name`.
`priority`	Debian platform only. The relative priority of the program for start and shutdown ordering. May be an integer or a Hash. An integer is used to define the start run levels; stop run levels are then 100-integer. A Hash is used to define values for specific run levels. For example, `{ 2 => [:start, 20], 3 => [:stop, 55] }` will set a priority of twenty for run level two and a priority of fifty-five for run level three.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`reload_command`	The command used to tell a service to reload its configuration.
`restart_command`	The command used to restart a service.

Attribute	Description
`service_name`	The name of the service. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`start_command`	The command used to start a service.
`status_command`	The command used to check the run status for a service.
`stop_command`	The command used to stop a service.
`supports`	A list of attributes that controls how the chef-client will attempt to manage a service: `:restart`, `:reload`, `:status`. For `:restart`, the init script or other service provider can use a restart command; if `:restart` is not specified, the chef-client will attempt to stop and then start a service. For `:reload`, the init script or other service provider can use a reload command. For `:status`, the init script or other service provider can use a status command to determine if the service is running; if `:status` is not specified, the chef-client will attempt to match the `service_name` against the process table as a regular expression, unless a pattern is specified as a parameter attribute. Default value: `{ :restart => false, :reload => false, :status => false }` for all platforms (except for the Red Hat platform family, which defaults to `{ :restart => false, :reload => false, :status => true }`.)

Providers

The **service** resource does not have service-specific short names. This is because the chef-client identifies the platform at the start of every chef-client run based on data collected by Ohai. The chef-client looks up the platform in the provider_mapping.rb file, and then determines the correct provider for that platform. In certain situations, such as when more than one init system is available on a node, a specific provider may need to be identified by using the `provider` attribute and the long name for that provider.

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Service::Init`	`service`	When this short name is used, the chef-client will determine the correct provider during the chef-client run.
`Chef::Provider::Service::Init::Debian`	`service`	The provider that is used with the Debian and Ubuntu platforms.
`Chef::Provider::Service::Upstart`	`service`	The provider that is used when Upstart is available on the platform.
`Chef::Provider::Service::Init::Freebsd`	`service`	The provider that is used with the FreeBSD platform.
`Chef::Provider::Service::Init::Gentoo`	`service`	The provider that is used with the Gentoo platform.
`Chef::Provider::Service::Init::Redhat`	`service`	The provider that is used with the Red Hat and CentOS platforms.
`Chef::Provider::Service::Solaris`	`service`	The provider that is used with the Solaris platform.
`Chef::Provider::Service::Windows`	`service`	The provider that is used with the Microsoft Windows platform.
`Chef::Provider::Service::Macosx`	`service`	The provider that is used with the Mac OS X platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Start a service

```
service "example_service" do
  action :start
end
```

Start a service, enable it

```
service "example_service" do
  supports :status => true, :restart => true, :reload => true
  action [ :enable, :start ]
end
```

Use a pattern

```
service "samba" do
  pattern "smbd"
  action [:enable, :start]
end
```

Manage a service, depending on the node platform

```
service "example_service" do
  case node["platform"]
  when "centos","redhat","fedora"
    service_name "redhat_name"
  else
    service_name "other_name"
  end
  supports :restart => true
  action [ :enable, :start ]
end
```

Change a service provider, depending on the node platform

```
service "example_service" do
  case node["platform"]
  when "ubuntu"
    if node["platform_version"].to_f >= 9.10
      provider Chef::Provider::Service::Upstart
    end
  end
  action [:enable, :start]
end
```

Set an IP address using variables and a template

The following example shows how the **template** resource can be used in a recipe to combine settings stored in an attributes file, variables within a recipe, and a template to set the IP addresses that are used by the Nginx service. The attributes file contains the following:

```
default['nginx']['dir'] = "/etc/nginx"
```

The recipe then does the following to:

- Declare two variables at the beginning of the recipe, one for the remote IP address and the other for the authorized IP address
- Use the **service** resource to restart and reload the Nginx service
- Load a template named `authorized_ip.erb` from the `/templates` directory that is used to set the IP address values based on the variables specified in the recipe

```
node.default['nginx']['remote_ip_var'] = "remote_addr"
node.default['nginx']['authorized_ips'] = ["127.0.0.1/32"]

service "nginx" do
  supports :status => true, :restart => true, :reload => true
end

template "authorized_ip" do
  path "#{node['nginx']['dir']}/authorized_ip"
  source "modules/authorized_ip.erb"
  owner 'root'
  group 'root'
  mode '0644'
  variables(
    :remote_ip_var => node['nginx']['remote_ip_var'],
    :authorized_ips => node['nginx']['authorized_ips']
  )

  notifies :reload, "service[nginx]", :immediately
end
```

where the `variables` attribute tells the template to use the variables set at the beginning of the recipe and the `source` attribute is used to call a template file located in the cookbook's `/templates` directory. The template file looks something like:

```
geo $<%= @remote_ip_var %> $authorized_ip {
  default no;
  <% @authorized_ips.each do |ip| %>
  <%= "#{ip} yes;" %>
  <% end %>
}
```

Use a cron timer to manage a service

The following example shows how to install the crond application using two resources and a variable:

```
#  the following code sample comes from the ``cron`` cookbook: https://github.com/opscode-cookbooks/cron

cron_package = case node['platform']
  when "redhat", "centos", "scientific", "fedora", "amazon"
    node['platform_version'].to_f >= 6.0 ? "cronie" : "vixie-cron"
  else
    "cron"
  end
```

```
package cron_package do
  action :install
end

service "crond" do
  case node['platform']
  when "redhat", "centos", "scientific", "fedora", "amazon"
    service_name "crond"
  when "debian", "ubuntu", "suse"
    service_name "cron"
  end
  action [:start, :enable]
end
```

where

- `cron_package` is a variable that is used to identify which platforms apply to which install packages
- the **package** resource uses the `cron_package` variable to determine how to install the crond application on various nodes (with various platforms)
- the **service** resource enables the crond application on nodes that have Red Hat, CentOS, Red Hat Enterprise Linux, Fedora, or Amazon Web Services, and the cron service on nodes that run Debian, Ubuntu, or openSUSE.

Restart a service, and then notify a different service

The following example shows how start a service named `example_service` and immediately notify the Nginx service to restart.

```
service "example_service" do
  action :start
  provider Chef::Provider::Service::Init
  notifies :restart, "service[nginx]", :immediately
end
```

where by using the default `provider` for the **service**, the recipe is telling the chef-client to determine the specific provider to be used during the chef-client run based on the platform of the node on which the recipe will run.

Stop a service, do stuff, and then restart it

The following example shows how to use the **execute**, **service**, and **mount** resources together to ensure that a node running on Amazon EC2 is running MySQL. This example does the following:

- Checks to see if the Amazon EC2 node has MySQL
- If the node has MySQL, stops MySQL
- Installs MySQL
- Mounts the node
- Restarts MySQL

```
#  the following code sample comes from the ``server_ec2`` recipe in the following cookbook: https://github.

if (node.attribute?('ec2') && ! FileTest.directory?(node['mysql']['ec2_path']))

  service "mysql" do
    action :stop
  end

  execute "install-mysql" do
    command "mv #{node['mysql']['data_dir']} #{node['mysql']['ec2_path']}"
    not_if do FileTest.directory?(node['mysql']['ec2_path']) end
  end

  [node['mysql']['ec2_path'], node['mysql']['data_dir']].each do |dir|
    directory dir do
      owner 'mysql'
      group 'mysql'
    end
  end

  mount node['mysql']['data_dir'] do
    device node['mysql']['ec2_path']
    fstype "none"
    options "bind,rw"
    action [:mount, :enable]
  end

  service "mysql" do
    action :start
  end

end
```

where

- the two **service** resources are used to stop, and then restart the MySQL service
- the **execute** resource is used to install MySQL

- the **mount** resource is used to mount the node and enable MySQL

Control a service using the execute resource

Warning

This is an example of something that should NOT be done. Use the **service** resource to control a service, not the **execute** resource.

Do something like this:

```
service "tomcat" do
  action :start
end
```

and NOT something like this:

```
execute "start-tomcat" do
  command "/etc/init.d/tomcat6 start"
  action :run
end
```

There is no reason to use the **execute** resource to control a service because the **service** resource exposes the `start_command` attribute directly, which gives a recipe full control over the command issued in a much cleaner, more direct manner.

smartos_package

Use the **smartos_package** resource to manage packages for the SmartOS platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **smartos_package** resource in a recipe is as follows:

```
smartos_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `smartos_package` tells the chef-client to use the `Chef::Provider::Package::Smartos` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)

Attribute	Description
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Smartos`	`smartos_package`	The provider that is used with the SmartOS platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
smartos_package "name of package" do
  action :install
end
```

solaris_package

The **solaris_package** resource is used to manage packages for the Solaris platform.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **solaris_package** resource in a recipe is as follows:

```
solaris_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `solaris_package` tells the chef-client to use the `Chef::Provider::Package::Solaris` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`options`	One (or more) additional options that are passed to the command.
`package_name`	The name of the package. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`response_file`	Optional. The direct path to the file used to pre-seed a package.
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Solaris`	`solaris_package`	The provider that is used with the Solaris platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
solaris_package "name of package" do
  action :install
end
```

subversion

Use the **subversion** resource to manage source control resources that exist in a Subversion repository.

Note

This resource is often used in conjunction with the **deploy** resource.

Syntax

The syntax for using the **subversion** resource in a recipe is as follows:

```
subversion "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `subversion` tells the chef-client to use the `Chef::Provider::Subversion` provider during the chef-client run.
- `"name"` is the location in which the source files will be placed and/or synchronized with the files under source control management
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

For example:

```
subversion "CouchDB Edge" do
  repository "http://svn.apache.org/repos/asf/couchdb/trunk"
  revision "HEAD"
  destination "/opt/mysources/couch"
  action :sync
end
```

where

- the name of the resource is `CouchDB Edge`

- the `repository` and `reference` nodes tell the chef-client which repository and revision to use

Actions

This resource has the following actions:

Action	Description
`:sync`	Default. Use to update the source to the specified version, or to get a new clone or checkout.
`:checkout`	Use to clone or check out the source. When a checkout is available, this provider does nothing.
`:export`	Use to export the source, excluding or removing any version control artifacts.
`:force_export`	Use to export the source, excluding or removing any version control artifacts and to force an export of the source that is overwriting the existing copy (if it exists).

Attributes

This resource has the following attributes:

Attribute	Description
`destination`	The path to the location to which the source will be cloned, checked out, or exported. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`group`	The system group that is responsible for the checked-out code.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`repository`	The URI for the Subversion repository.
`revision`	The revision to be checked out. This can be symbolic, like `HEAD` or it can be a source control management-specific revision identifier. Default value: `HEAD`.
`svn_arguments`	The extra arguments that are passed to the Subversion command.
`svn_info_args`	Use when the `svn info` command is used by the chef-client and arguments need to be passed. (The `svn_arguments` command does not work when the `svn info` command is used.)
`svn_password`	The password for the user that has access to the Subversion repository.
`svn_username`	The user name for a user that has access to the Subversion repository.
`timeout`	The amount of time (in seconds) to wait for a command to execute before timing out. When this attribute is specified using the **deploy** resource, the value of the `timeout` attribute is passed from the **deploy** resource to the **subversion** resource.
`user`	The system user that is responsible for the checked-out code.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Subversion`	`subversion`	This provider work only with Subversion.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Get the latest version of an application

```
subversion "CouchDB Edge" do
  repository "http://svn.apache.org/repos/asf/couchdb/trunk"
  revision "HEAD"
  destination "/opt/mysources/couch"
  action :sync
end
```

template

Use the **template** resource to manage the contents of a file using an Embedded Ruby (ERB) template by transferring files from a sub-directory of `COOKBOOK_NAME/templates/default` to a specified path located on a host that is running the chef-client. This resource includes actions and attributes from the **file** resource. Template files managed by the **template** resource follow the same file specificity rules as the **remote_file** and **file** resources.

To use a template, two things must happen:

1. A template resource must be added to a recipe
2. An Embedded Ruby (ERB) template must be added to a cookbook

For example, the following template file and template resource settings can be used to manage a configuration file named `/etc/sudoers`. Within a cookbook that uses sudo, the following resource could be added to `/recipes/default.rb`:

```
template "/etc/sudoers" do
  source "sudoers.erb"
  mode '0440'
  owner 'root'
  group 'root'
  variables({
     :sudoers_groups => node[:authorization][:sudo][:groups],
     :sudoers_users => node[:authorization][:sudo][:users]
  })
end
```

And then create a template called `sudoers.erb` and save it to `templates/default/sudoers.erb`:

```
#
# /etc/sudoers
#
# Generated by Chef for <%= node[:fqdn] %>
#

Defaults        !lecture,tty_tickets,!fqdn

# User privilege specification
root          ALL=(ALL) ALL

<% @sudoers_users.each do |user| -%>
<%= user %>   ALL=(ALL) <%= "NOPASSWD:" if @passwordless %>ALL
<% end -%>

# Members of the sysadmin group may gain root privileges
%sysadmin     ALL=(ALL) <%= "NOPASSWD:" if @passwordless %>ALL

<% @sudoers_groups.each do |group| -%>
# Members of the group '<%= group %>' may gain root privileges
%<%= group %> ALL=(ALL) <%= "NOPASSWD:" if @passwordless %>ALL
<% end -%>
```

And then set the default attributes in `attributes/default.rb`:

```
default["authorization"]["sudo"]["groups"] = [ "sysadmin","wheel","admin" ]
default["authorization"]["sudo"]["users"]  = [ "jerry","greg"]
```

Syntax

The syntax for using the **template** resource in a recipe is as follows:

```
template "name" do
  source "template_name.erb"
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `template` tells the chef-client to use the `Chef::Provider::File::Template` provider during the chef-client run
- `name` is the path to the location in which a file will be created and the name of the file to be managed; for example: `/var/www/html/index.html`, where `/var/www/html/` is the fully qualified path to the location and `index.html` is the name of the file
- `source` is the template file that will be used to create the file on the node, for example: `index.html.erb`; the template file is located in the `/templates` directory of a cookbook
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description

Action	Description
`:create`	Default. Use to create a file. If a file already exists (but does not match), use to update that file to match.
`:create_if_missing`	Use to create a file only if the file does not exist. (When the file exists, nothing happens.)
`:delete`	Use to delete a file.
`:touch`	Use to touch a file. This updates the access (atime) and file modification (mtime) times for a file. (This action may be used with this resource, but is typically only used with the **file** resource.)

Attributes

This resource has the following attributes:

Attribute	Description
`atomic_update`	Use to perform atomic file updates on a per-resource basis. Set to `true` for atomic file updates. Set to `false` for non-atomic file updates. (This setting overrides `file_atomic_update`, which is a global setting found in the client.rb file.) Default value: `true`.
`backup`	The number of backups to be kept. Set to `false` to prevent backups from being kept. Default value: `5`.
`cookbook`	The cookbook in which a file is located (if it is not located in the current cookbook). The default value is the current cookbook.
`force_unlink`	Use to specify how the chef-client handles certain situations when the target file turns out not to be a file. For example, when a target file is actually a symlink. Set to `true` to have the chef-client delete the non-file target and replace it with the specified file. Set to `false` for the chef-client to raise an error. Default value: `false`.
`group`	A string or ID that identifies the group owner by group name, including fully qualified group names such as `domain\group` or `group@domain`. If this value is not specified, existing groups will remain unchanged and new group assignments will use the default `POSIX` group (if available).
`helper`	Use to define a helper method inline. For example: `helper(:hello_world) { "hello world" }` or `helper(:app) { node["app"] }` or `helper(:app_conf) { |setting| node["app"][setting] }`. Default value: `{}`.
`helpers`	Use to define a helper module inline or in a library. For example, an inline module: `helpers do`, which is then followed by a block of Ruby code. And for a library module: `helpers(MyHelperModule)`. Default value: `[]`.
`inherits`	Microsoft Windows only. Use to specify that a file inherits rights from its parent directory. Default value: `true`.
`local`	Use to load a template from a local path. By default, the chef-client loads templates from a cookbook's `/templates` directory. When this attribute is set to `true`, use the `source` attribute specify the path to a template on the local node. Default value: `false`.
`manage_symlink_source`	Use to have the chef-client detect and manage the source file for a symlink. Possible values: `nil`, `true`, or `false`. When this value is set to `nil`, the chef-client will manage a symlink's source file and emit a warning. When this value is set to `true`, the chef-client will manage a symlink's source file and not emit a warning. Default value: `nil`. The default value will be changed to `false` in a future version.
`mode`	A quoted string that defines the octal mode for a file. If `mode` is not specified and if the file already exists, the existing mode on the file is used. If `mode` is not specified, the file does not exist, and the `:create` action is specified, the chef-client will assume a mask value of `"0777"` and then apply the umask for the system on which the file will be created to the `mask` value. For example, if the umask on a system is `"022"`, the chef-client would use the default value of `"0755"`. The behavior is different depending on the platform. UNIX- and Linux-based systems: A quoted string that defines the octal mode that is passed to chmod. If the value is specified as a quoted string, it will work exactly as if the `chmod` command was passed. If the value is specified as an integer, prepend a zero (`0`) to the value to ensure it is interpreted as an octal number. For example, to assign read, write, and execute rights for all users, use `"0777"` or `"777"`; for the same rights, plus the sticky bit, use `"01777"` or `"1777"`. Microsoft Windows: A quoted string that defines the octal mode that is translated into rights for Microsoft Windows security. Values up to `"0777"` are allowed (no sticky bits) and mean the same in Microsoft Windows as they do in UNIX, where `4` equals `GENERIC_READ`, `2` equals `GENERIC_WRITE`, and `1` equals `GENERIC_EXECUTE`. This attribute cannot be used to set `:full_control`. This attribute has no effect if not specified, but when this attribute and `rights` are both specified, the effects will be cumulative.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::File::Template`	`template`	The default provider for all platforms.

File Specificity

A cookbook is frequently designed to work across many platforms and is often required to distribute a specific template to a specific platform. A cookbook can be designed to support the distribution of templates across platforms, while ensuring that the correct template ends up on each system.

Pattern

The pattern for file specificity is as follows:

1. host-node[:fqdn]
2. node[:platform]-node[:platform_version]
3. node[:platform]-version_components: The version string is split on decimals and searched from greatest specificity to least; for example, if the location from the last rule was centos-5.7.1, then centos-5.7 and centos-5 would also be searched.
4. node[:platform]
5. default

Example

A cookbook may have a `/templates` directory structure like this:

```
templates/
  windows-6.2
  windows-6.1
  windows-6.0
  windows
  default
```

and a resource that looks something like the following:

```
template "C:\path\to\file\text_file.txt" do
  source "text_file.txt"
  mode '0755'
  owner 'root'
  group 'root'
end
```

This resource would be matched in the same order as the `/templates` directory structure. For a node named “host-node-desktop” that is running Windows 7, the second item would be the matching item and the location:

```
/templates
  windows-6.2/text_file.txt
  windows-6.1/text_file.txt
  windows-6.0/text_file.txt
  windows/text_file.txt
  default/text_file.txt
```

Helpers

A helper is a method or a module that can be used to extend a template. There are three approaches:

- An inline helper method
- An inline helper module
- A cookbook library module

Use the `helper` attribute in a recipe to define an inline helper method. Use the `helpers` attribute to define an inline helper module or a cookbook library module.

Inline Methods

A template helper method is always defined inline on a per-resource basis. A simple example:

```
template "/path" do
  helper(:hello_world) { "hello world" }
end
```

Another way to define an inline helper method is to reference a node object so that repeated calls to one (or more) cookbook attributes can be done efficiently:

```
template "/path" do
```

```
  helper(:app) { node["app"] }
end
```

An inline helper method can also take arguments:

```
template "/path" do
  helper(:app_conf) { |setting| node["app"][setting] }
end
```

Once declared, a template can then use the helper methods to build a file. For example:

```
Say hello: <%= hello_world %>
```

Or:

```
node["app"]["listen_port"] is: <%= app["listen_port"] %>
```

Or:

```
node["app"]["log_location"] is: <%= app_conf("log_location") %>
```

Inline Modules

A template helper module can be defined inline on a per-resource basis. This approach can be useful when a template requires more complex information. For example:

```
template "/path" do
  helpers do

    def hello_world
      "hello world"
    end

    def app
      node["app"]
    end

    def app_conf(setting)
      node["app"][setting]
    end

  end
end
```

where the `hello_world`, `app`, and `app_conf(setting)` methods comprise the module that extends a template.

Library Modules

A template helper module can be defined in a library. This is useful when extensions need to be reused across recipes or to make it easier to manage code that would otherwise be defined inline on a per-recipe basis.

```
template "/path/to/template.erb" do
  helpers(MyHelperModule)
end
```

Host Notation

The naming of folders within cookbook directories must literally match the host notation used for template specificity matching. For example, if a host is named `foo.example.com`, then the folder must be named `host-foo.example.com`.

Partial Templates

A template can be built in a way that allows it to contain references to one (or more) smaller template files. (These smaller template files are also referred to as partials.) A partial can be referenced from a template file in one of the following ways:

- By using the Ruby `render` method in the template file
- By using the **template** resource and the `variables` parameter.

render Method

Use the `render` method in a template to reference a partial template file:

```
<%= render "partial_name.txt.erb", :option => {} %>
```

where `partial_name` is the name of the partial template file and `:option` is one (or more) of the following:

Option	Description
`:cookbook`	By default, a partial template file is assumed to be located in the cookbook that contains the top-level template. Use this

Option	Description
	option to specify the path to a different cookbook
`:local`	Indicates that the name of the partial template file should be interpreted as a path to a file in the local file system or looked up in a cookbook using the normal rules for template files. Set to `true` to interpret as a path to a file in the local file system and to `false` to use the normal rules for template files
`:source`	By default, a partial template file is identified by its file name. Use this option to specify a different name or a local path to use (instead of the name of the partial template file)
`:variables`	A hash of `variable_name => value` that will be made available to the partial template file. When this option is used, any variables that are defined in the top-level template that are required by the partial template file must have them defined explicitly using this option

For example:

```
<%= render "simple.txt.erb", :variables => {:user => Etc.getlogin }, :local => true %>
```

Transfer Frequency

The chef-client caches a template when it is first requested. On each subsequent request for that template, the chef-client compares that request to the template located on the Chef server. If the templates are the same, no transfer occurs.

Variables

A template is an Embedded Ruby (ERB) template. An Embedded Ruby (ERB) template allows Ruby code to be embedded inside a text file within specially formatted tags. Ruby code can be embedded using expressions and statements. An expression is delimited by `<%=` and `%>`. For example:

```
``<%= "my name is #{$ruby}" %>``
```

A statement is delimited by a modifier, such as `if`, `elseif`, and `else`. For example:

```
if false
   # this won't happen
elsif nil
   # this won't either
else
   # code here will run though
end
```

Using a Ruby expression is the most common approach for defining template variables because this is how all variables that are sent to a template are referenced. Whenever a template needs to use an `each`, `if`, or `end`, use a Ruby statement.

When a template is rendered, Ruby expressions and statements are evaluated by the chef-client. The variables listed in the resource's variables parameter and the node object are evaluated. The chef-client then passes these variables to the template, where they will be accessible as instance variables within the template; the node object can be accessed just as if it were part of a recipe, using the same syntax.

For example, a simple template resource like this:

```
node[:fqdn] = "latte"
template "/tmp/foo" do
  source 'foo.erb'
  variables({
    :x_men => "are keen"
  })
end
```

And a simple Embedded Ruby (ERB) template like this:

```
The node <%= node[:fqdn] %> thinks the x-men <%= @x_men %>
```

Would render something like:

```
The node latte thinks the x-men are keen
```

Even though this is a very simple example, the full capabilities of Ruby can be used to tackle even the most complex and demanding template requirements.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Configure a file from a template

```
template "/tmp/config.conf" do
  source "config.conf.erb"
end
```

Configure a file from a local template

```
template "/tmp/config.conf" do
  local true
  source "/tmp/config.conf.erb"
end
```

Configure a file using a variable map

```
template "/tmp/config.conf" do
  source "config.conf.erb"
  variables(
    :config_var => node["configs"]["config_var"]
  )
end
```

Use the ``not_if`` condition

The following example shows how to use the `not_if` condition to create a file based on a template and using the presence of an attribute on the node to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if { node[:some_value] }
end
```

The following example shows how to use the `not_if` condition to create a file based on a template and then Ruby code to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if do
    File.exists?("/etc/passwd")
  end
end
```

The following example shows how to use the `not_if` condition to create a file based on a template and using a Ruby block (with curly braces) to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if {File.exists?("/etc/passwd")}
end
```

The following example shows how to use the `not_if` condition to create a file based on a template and using a string to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  not_if "test -f /etc/passwd"
end
```

Use the ``only_if`` condition

The following example shows how to use the `only_if` condition to create a file based on a template and using the presence of an attribute on the node to specify the condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  only_if { node[:some_value] }
end
```

The following example shows how to use the `only_if` condition to create a file based on a template, and then use Ruby to specify a condition:

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  only_if do ! File.exists?("/etc/passwd") end
end
```

The following example shows how to use the `only_if` condition to create a file based on a template and using a string to specify the condition:

```
template "/tmp/somefile" do
```

```
  mode '0644'
  source "somefile.erb"
  only_if "test -f /etc/passwd"
end
```

Use a whitespace array (``%w``)

The following example shows how to use a Ruby whitespace array to define a list of configuration tools, and then use that list of tools within the **template** resource to ensure that all of these configuration tools are using the same RSA key:

```
%w{openssl.cnf pkitool vars Rakefile}.each do |f|
  template "/etc/openvpn/easy-rsa/#{f}" do
    source "#{f}.erb"
    owner 'root'
    group 'root'
    mode '0755'
  end
end
```

Use a relative path

```
template "#{ENV['HOME']}/chef-getting-started.txt" do
  source "chef-getting-started.txt.erb"
  mode '0644'
end
```

Delay notifications

```
template "/etc/nagios3/configures-nagios.conf" do
  # other parameters
  notifies :run, "execute[test-nagios-config]", :delayed
end
```

Notify immediately

By default, notifications are `:delayed`, that is they are queued up as they are triggered, and then executed at the very end of a chef-client run. To run an action immediately, use `:immediately`:

```
template "/etc/nagios3/configures-nagios.conf" do
  # other parameters
  notifies :run, "execute[test-nagios-config]", :immediately
end
```

and then the chef-client would immediately run the following:

```
execute "test-nagios-config" do
  command "nagios3 --verify-config"
  action :nothing
end
```

Notify multiple resources

```
template "/etc/chef/server.rb" do
  source "server.rb.erb"
  owner 'root'
  group 'root'
  mode '0644'
  notifies :restart, "service[chef-solr]", :delayed
  notifies :restart, "service[chef-solr-indexer]", :delayed
  notifies :restart, "service[chef-server]", :delayed
end
```

Reload a service

```
template "/tmp/somefile" do
  mode '0644'
  source "somefile.erb"
  notifies :reload, "service[apache]", :immediately
end
```

Restart a service when a template is modified

```
template "/etc/www/configures-apache.conf" do
  notifies :restart, "service[apache]", :immediately
end
```

Send notifications to multiple resources

To send notifications to multiple resources, just use multiple attributes. Multiple attributes will get sent to the notified resources in the order specified.

```
template "/etc/netatalk/netatalk.conf" do
```

```
  notifies :restart, "service[afpd]", :immediately
  notifies :restart, "service[cnid]", :immediately
end

service "afpd"
service "cnid"
```

Execute a command using a template

The following example shows how to set up IPv4 packet forwarding using the **execute** resource to run a command named `forward_ipv4` that uses a template defined by the **template** resource:

```
execute "forward_ipv4" do
  command "echo > /proc/.../ipv4/ip_forward"
  action :nothing
end

template "/etc/file_name.conf" do
  source "routing/file_name.conf.erb"
  notifies :run, 'execute[forward_ipv4]', :delayed
end
```

where the `command` attribute for the **execute** resource contains the command that is to be run and the `source` attribute for the **template** resource specifies which template to use. The `notifies` attribute for the **template** specifies that the `execute[forward_ipv4]` (which is defined by the **execute** resource) should be queued up and run at the end of the chef-client run.

Set an IP address using variables and a template

The following example shows how the **template** resource can be used in a recipe to combine settings stored in an attributes file, variables within a recipe, and a template to set the IP addresses that are used by the Nginx service. The attributes file contains the following:

```
default['nginx']['dir'] = "/etc/nginx"
```

The recipe then does the following to:

- Declare two variables at the beginning of the recipe, one for the remote IP address and the other for the authorized IP address
- Use the **service** resource to restart and reload the Nginx service
- Load a template named `authorized_ip.erb` from the `/templates` directory that is used to set the IP address values based on the variables specified in the recipe

```
node.default['nginx']['remote_ip_var'] = "remote_addr"
node.default['nginx']['authorized_ips'] = ["127.0.0.1/32"]

service "nginx" do
  supports :status => true, :restart => true, :reload => true
end

template "authorized_ip" do
  path "#{node['nginx']['dir']}/authorized_ip"
  source "modules/authorized_ip.erb"
  owner 'root'
  group 'root'
  mode '0644'
  variables(
    :remote_ip_var => node['nginx']['remote_ip_var'],
    :authorized_ips => node['nginx']['authorized_ips']
  )

  notifies :reload, "service[nginx]", :immediately
end
```

where the `variables` attribute tells the template to use the variables set at the beginning of the recipe and the `source` attribute is used to call a template file located in the cookbook's `/templates` directory. The template file looks something like:

```
geo $<%= @remote_ip_var %> $authorized_ip {
  default no;
  <% @authorized_ips.each do |ip| %>
  <%= "#{ip} yes;" %>
  <% end %>
}
```

Add a rule to an IP table

The following example shows how to add a rule named `test_rule` to an IP table using the **execute** resource to run a command using a template that is defined by the **template** resource:

```
execute 'test_rule' do
  command "command_to_run
    --option value
    ...
    --option value
    --source #{node[:name_of_node][:ipsec][:local][:subnet]}
    -j test_rule"
  action :nothing
end

template "/etc/file_name.local" do
  source "routing/file_name.local.erb"
  notifies :run, 'execute[test_rule]', :delayed
end
```

where the `command` attribute for the **execute** resource contains the command that is to be run and the `source` attribute for the **template** resource specifies which template to use. The `notifies` attribute for the **template** specifies that the `execute[test_rule]` (which is defined by the **execute** resource) should be queued up and run at the end of the chef-client run.

Apply proxy settings consistently across a Chef organization

The following example shows how a template can be used to apply consistent proxy settings for all nodes of the same type:

```
template "#{node[:matching_node][:dir]}/sites-available/site_proxy.conf" do
  source "site_proxy.matching_node.conf.erb"
  owner 'root'
  group 'root'
  mode '0644'
  variables(
    :ssl_certificate =>    "#{node[:matching_node][:dir]}/shared/certificates/site_proxy.crt",
    :ssl_key =>            "#{node[:matching_node][:dir]}/shared/certificates/site_proxy.key",
    :listen_port =>        node[:site][:matching_node_proxy][:listen_port],
    :server_name =>        node[:site][:matching_node_proxy][:server_name],
    :fqdn =>               node[:fqdn],
    :server_options =>     node[:site][:matching_node][:server][:options],
    :proxy_options =>      node[:site][:matching_node][:proxy][:options]
  )
end
```

where `matching_node` represents a type of node (like Nginx) and `site_proxy` represents the type of proxy being used for that type of node (like Nexus).

Get template settings from a local file

The **template** resource can be used to render a template based on settings contained in a local file on disk or to get the settings from a template in a cookbook. Most of the time, the settings are retrieved from a template in a cookbook. The following example shows how the **template** resource can be used to retrieve these settings from a local file.

The following example is based on a few assumptions:

- The environment is a Ruby on Rails application that needs render a file named `database.yml`
- Information about the application—the user, their password, the server—is stored in a data bag on the Chef server
- The application is already deployed to the system and that only requirement in this example is to render the `database.yml` file

The application source tree looks something like:

```
myapp/
-> config/
   -> database.yml.erb
```

Note

There should not be a file named `database.yml` (without the `.erb`), as the `database.yml` file is what will be rendered using the **template** resource.

The deployment of the app will end up in `/srv`, so the full path to this template would be something like `/srv/myapp/current/config/database.yml.erb`.

The content of the template itself may look like this:

```
<%= @rails_env %>:
   adapter: <%= @adapter %>
   host: <%= @host %>
   database: <%= @database %>
   username: <%= @username %>
   password: <%= @password %>
   encoding: 'utf8'
   reconnect: true
```

The recipe will be similar to the following:

```
results = search(:node, "role:myapp_database_master AND environment:#{node.chef_environment}")
db_master = results[0]

template "/srv/myapp/shared/database.yml" do
  source "/srv/myapp/current/config/database.yml.erb"
  local true
  variables(
    :rails_env => node.chef_environment,
    :adapter => db_master['myapp']['db_adapter'],
    :host => db_master['fqdn'],
    :database => "myapp_#{node.chef_environment}",
    :username => "myapp",
    :password => "SUPERSECRET",
  )
end
```

where:

- the `search` method in the Recipe DSL is used to find the first node that is the database master (of which there should only be one)
- the `:adapter` attribute may also require an attribute to have been set on a role, which then determines the correct adapter

The template will render similar to the following:

```
production:
  adapter: mysql
  host: domU-12-31-39-14-F1-C3.compute-1.internal
  database: myapp_production
  username: myapp
  password: SUPERSECRET
  encoding: utf8
  reconnect: true
```

This example showed how to use the **template** resource to render a template based on settings contained in a local file. Some other issues that should be considered when using this type of approach include:

- Should the `database.yml` file be in a `.gitignore` file?
- How do developers run the application locally?
- How does this work with chef-solo?

user

Use the **user** resource to add users, update existing users, remove users, and to lock/unlock user passwords.

Note

System attributes are collected by Ohai at the start of every chef-client run. By design, the actions available to the **user** resource are processed **after** the start of the chef-client run. This means that attributes added or modified by the **user** resource during the chef-client run must be reloaded before they can be available to the chef-client. These attributes can be reloaded in two ways: by picking up the values at the start of the (next) chef-client run or by using the ohai resource to reload these attributes during the current chef-client run.

Syntax

The syntax for using the **user** resource in a recipe is as follows:

```
user "name" do
  attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `user` tells the chef-client to use one of the following providers during the chef-client run: `Chef::Provider::User::Useradd`, `Chef::Provider::User::Pw`, `Chef::Provider::User::Dscl`, or `Chef::Provider::User::Windows`. The provider that is used by the chef-client depends on the platform of the machine on which the chef-client run is taking place
- `name` is the name of the resource block; when the `username` attribute is not specified as part of a recipe, `name` is also the name of the user
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:create`	Default. Use to create a user with given attributes. If a user already exists (but does not match), use to update that user to match.
`:remove`	Use to remove a user.
`:modify`	Use to modify an existing user. This action will raise an exception if the user does not exist.
`:manage`	Use to manage an existing user. This action will do nothing if the user does not exist.
`:lock`	Use to lock a user's password.

Action	Description
`:unlock`	Use to unlock a user's password.

Attributes

This resource has the following attributes:

Attribute	Description
`comment`	One (or more) comments about the user.
`gid`	The identifier for the group.
`home`	The location of the home directory.
`password`	The password shadow hash. This attribute requires that `ruby-shadow` be installed. This is part of the Debian package: `libshadow-ruby1.8`.
`provider`	Optional. Use to explicitly specify a provider. (See "Providers" section below for more information.)
`shell`	The login shell.
`supports`	A Mash where keys represent features and values are booleans that indicate if that feature is supported. Default value: `:manage_home => false, :non_unique => false`.
`system`	Use to create a system user. This attribute may be used with `useradd` as the provider to create a system user which passes the `-r` flag to `useradd`.
`uid`	The numeric user identifier.
`username`	The name of the user. Default value: the `name` of the resource block. (See "Syntax" section above for more information.)

Supported Features

The `supports` attribute allows a list of supported features to be identified. There are two features of note:

- `:manage_home` indicates whether a user's home directory will be created when the user is created. When the `Useradd` provider is used, `-dm` wil be passed to `useradd` (when the `:create` action is used) and `-d` will be passed to `usermod` (when the `:manage` or `:modify` actions are used). If `supports :manage_home=>true`, the **user** resource passes the `-d` and `-m` parameters together (i.e. `-dm`) to `usermod`.

  When the `Windows` provider is used, Microsoft Windows does not create a home directory for a user until that user logs on for the first time; specifying the home directory does not have any effect as to where Microsoft Windows ultimately places the home directory.
- `:non_unique` indicates whether non-unique UIDs are allowed. This option is currently unused by the existing providers.

Password Shadow Hash

There are a number of encryption options and tools that can be used to create a password shadow hash. In general, using a strong encryption method like SHA-512 and the `passwd` command in the OpenSSL toolkit is a good approach, however the encryption options and tools that are available may be different from one distribution to another. The following examples show how the command line can be used to create a password shadow hash. When using the `passwd` command in the OpenSSL tool:

```
openssl passwd -1 "theplaintextpassword"
```

When using `mkpasswd`:

```
mkpasswd -m sha-512
```

For more information:

- http://www.openssl.org/docs/apps/passwd.html
- Check the local documentation or package repository for the distribution that is being used. For example, on Ubuntu 9.10-10.04, the `mkpasswd` package is required and on Ubuntu 10.10+ the `whois` package is required.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::User::Useradd`	`user`	The default provider for the **user** resource.

Long name	Short name	Notes
`Chef::Provider::User::Pw`	`user`	The provider that is used with the FreeBSD platform.
`Chef::Provider::User::Dscl`	`user`	The provider that is used with the Mac OS X platform.
`Chef::Provider::User::Windows`	`user`	The provider that is used with all Microsoft Windows platforms.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Create a random user

```
user "random" do
  supports :manage_home => true
  comment "Random User"
  uid 1234
  gid "users"
  home "/home/random"
  shell "/bin/bash"
  password "$1$JJsvHslV$szsCjVEroftprNn4JHtDi."
end
```

Create a system user

```
user "systemguy" do
  comment "system guy"
  system true
  shell "/bin/false"
end
```

Create a system user with a variable

The following example shows how to create a system user using a variable called `user_home` where the matching nodes have a group identifier that is the same as the node, and the login shell is `/bin/bash`:

```
user_home = "/#{node[:matching_node][:user]}"

user node[:matching_node][:group] do
  gid node[:matching_node][:group]
  shell "/bin/bash"
  home user_home
  system true
  action :create
end
```

where `matching_node` represents a type of node. For example, if the `user_home` variable specified `{node[:nginx]...}`, a recipe might look something like this:

```
user_home = "/#{node[:nginx][:user]}"

user node[:nginx][:group] do
  gid node[:nginx][:group]
  shell "/bin/bash"
  home user_home
  system true
  action :create
end
```

windows_package

Use the **windows_package** resource to manage Microsoft Installer Package (MSI) packages for the Microsoft Windows platform.

Note

This resource effectively replaces the `windows_package` resource found in the **windows** cookbook by moving that functionality into the chef-client. The **windows** cookbook may still be used, but in that situation use the generic **package** resource instead of the **windows_package** resource.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **windows_package** resource in a recipe is as follows:

```
windows_package "name" do
  some_attribute "value" # see attributes section below
  ...
  action :action # see actions section below
end
```

where

- `windows_package` tells the chef-client to use the `Chef::Provider::Package::Windows` provider during the chef-client run
- `"name"` is the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:remove`	Use to remove a package.

Attributes

This resource has the following attributes:

Attribute	Description
`installer_type`	The package type. Possible values: `:msi`.
`options`	One (or more) additional options that are passed to the command.
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`returns`	A comma-delimited list of return codes, which indicate the success or failure for the command that was run remotely. This code signals a successful `:install` action. Default value: `0`.
`source`	Optional. The package source for providers that use a local file. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`timeout`	The amount of time (in seconds) to wait before timing out. Default value: `600` (seconds).

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Windows`	`windows_package`	The provider that is used with the Microsoft Windows platform.

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install a package

```
windows_package "7zip" do
  action :install
  source 'C:\7z920.msi'
end
```

yum_package

Use the **yum_package** resource to install, upgrade, and remove packages with Yum for the Red Hat and CentOS platforms. The

yum_package resource is able to resolve `provides` data for packages much like Yum can do when it is run from the command line. This allows a variety of options for installing packages, like minimum versions, virtual provides, and library names.

Note

Support for using file names to install packages (as in `yum_package "/bin/sh"`) is not available because the volume of data required to parse for this is excessive.

Note

In many cases, it is better to use the **package** resource instead of this one. This is because when the **package** resource is used in a recipe, the chef-client will use details that are collected by Ohai at the start of the chef-client run to determine the correct package application. Using the **package** resource allows a recipe to be authored in a way that allows it to be used across many platforms. That said, there are scenarios where using an application-specific package is preferred.

Syntax

The syntax for using the **yum_package** resource in a recipe is as follows:

```
yum_package "name" do
  attribute "value" # see attributes section below
  ...
  action :action
end
```

where

- `yum_package` tells the chef-client to use the `Chef::Provider::Package::Yum` provider during the chef-client run
- `name` is the name of the resource block; when the `package_name` attribute is not specified as part of a recipe, `name` is also the name of the package
- `attribute` is zero (or more) of the attributes that are available for this resource
- `:action` identifies which steps the chef-client will take to bring the node into the desired state

Actions

This resource has the following actions:

Action	Description
`:install`	Default. Use to install a package. If a version is specified, use to install the specified version of a package.
`:upgrade`	Use to install a package and/or to ensure that a package is the latest version.
`:remove`	Use to remove a package.
`:purge`	Use to purge a package. This action typically removes the configuration files as well as the package.

Attributes

This resource has the following attributes:

Attribute	Description
`allow_downgrade`	Use to downgrade a package to satisfy requested version requirements.
`arch`	The architecture of the package that will be installed or upgraded. (This value can also be passed as part of the package name.)
`flush_cache`	Yum automatically synchronizes remote metadata to a local cache. The chef-client creates a copy of the local cache, and then stores it in-memory during the chef-client run. The in-memory cache allows packages to be installed during the chef-client run without the need to continue synchronizing the remote metadata to the local cache while the chef-client run is in-progress. Use this attribute to flush the in-memory cache before or after a Yum operation that installs, upgrades, or removes a package. Default value: `{ :before => false, :after => false }`. **Note** The `flush_cache` attribute does not flush the local Yum cache! Use Yum tools—`yum clean headers`, `yum clean packages`, `yum clean all`—to clean the local Yum cache.
`options`	One (or more) additional options that are passed to the command.

Attribute	Description
`package_name`	One of the following: the name of a package, the name of a package and its architecture; the name of a dependency. Default value: the `name` of the resource block. (See “Syntax” section above for more information.)
`provider`	Optional. Use to explicitly specify a provider. (See “Providers” section below for more information.)
`source`	Optional. The package source for providers that use a local file.
`version`	The version of a package to be installed or upgraded.

Providers

This resource has the following providers:

Long name	Short name	Notes
`Chef::Provider::Package`	`package`	When this short name is used, the chef-client will attempt to determine the correct provider during the chef-client run.
`Chef::Provider::Package::Yum`	`yum_package`	

Examples

The following examples demonstrate various approaches for using resources in recipes. If you want to see examples of how Chef uses resources in recipes, take a closer look at the cookbooks that Chef authors and maintains: https://github.com/opscode-cookbooks.

Install an exact version

```
yum_package "netpbm = 10.35.58-8.el5"
```

Install a minimum version

```
yum_package "netpbm >= 10.35.58-8.el5"
```

Install a minimum version using the default action

```
yum_package "netpbm"
```

To install a package

```
yum_package "netpbm" do
  action :install
end
```

To install a partial minimum version

```
yum_package "netpbm >= 10"
```

To install a specific architecture

```
yum_package "netpbm" do
  arch "i386"
end
```

or:

```
yum_package "netpbm.x86_64"
```

To install a specific version-release

```
yum_package "netpbm" do
  version "10.35.58-8.el5"
end
```

To install a specific version (even when older than the current)

```
yum_package "tzdata" do
  version "2011b-1.el5"
  allow_downgrade true
end
```

Handle cookbook_file and yum_package resources in the same recipe

When a **cookbook_file** resource and a **yum_package** resource are both called from within the same recipe, use the `flush_cache` attribute to dump the in-memory Yum cache, and then use the repository immediately to ensure that the correct package is installed:

```
cookbook_file "/etc/yum.repos.d/custom.repo" do
  source "custom"
  mode '0644'
end

yum_package "only-in-custom-repo" do
  action :install
  flush_cache [:before]
end
```