ChefSpec

ChefSpec is a unit testing framework for testing Chef cookbooks. ChefSpec makes it easy to write examples and get fast feedback on cookbook changes without the need for virtual machines or cloud servers.

ChefSpec runs your cookbook locally using Chef Solo without actually converging a node. This has two primary benefits:

- It's really fast!
- Your tests can vary node attributes, operating systems, and search results to assert behavior under varying conditions.

What people are saying

> *I just wanted to drop you a line to say "HELL YES!" to ChefSpec. - Joe Goggins*

> *OK chefspec is my new best friend. Delightful few hours working with it. - Michael Ivey*

Chat with us - #chefspec on Freenode

Important Notes

- **ChefSpec 3 requires Chef 11+! Please use the 2.x series for Chef 9 & 10 compatibility.**
- **This documentation corresponds to the master branch, which may be unreleased. Please check the README of the latest git tag or the gem's source for your version' documentation!**
- **Each resource matcher is self-documented using Yard and has a**

corresponding aruba test from the examples directory.

- **ChefSpec 3.0 requires Ruby 1.9 or higher!**

If you are migrating from ChefSpec v2.0.0, you should require the deprecations module after requiring `chefspec`:

```
# spec_helper.rb
require 'chefspec'
require 'chefspec/deprecations'
```

After you have converted your specs, you can safely remove the deprecations module.

Writing a Cookbook Example

If you want `knife` to automatically generate spec stubs for you, install knife-spec.

Given an extremely basic Chef recipe that just installs an operating system package:

```
package 'foo'
```

the associated ChefSpec test might look like:

```
require 'chefspec'

describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new.converge(described_recipe) }

  it 'installs foo' do
    expect(chef_run).to install_package('foo')
  end
end
```

Let's step through this file to see what is happening:

1. At the top of the spec file we require the chefspec gem. This is required so that our custom matchers are loaded. In larger projects, it is common practice to create a file named "spec_helper.rb" and include ChefSpec and perform other setup tasks in that file.
2. The `describe` keyword is part of RSpec and indicates that everything nested beneath is describing the `example::default` recipe. The convention is to have a separate spec for each recipe in your cookbook.
3. The `let` block on creates the `ChefSpec:Runner` and then does a fake Chef run with the run_list of `example::default`. Any subsequent examples can then refer to `chef_run` in order to make assertions about the resources that were created during the mock converge.
4. The `described_recipe` macro is a ChefSpec helper method that infers the recipe from the `describe` block. Alternatively you could specify the recipe directly.
5. The `it` block is an example specifying that the `foo` package is installed. Normally you will have multiple `it` blocks per recipe, each making a single assertion.

Configuration

ChefSpec exposes a configuration layer at the global level and at the `Runner` level. The following settings are available:

```
RSpec.configure do |config|
  # Specify the path for Chef Solo to find cookbooks (default: [infe
rred from
  # the location of the calling spec file])
  config.cookbook_path = '/var/cookbooks'

  # Specify the path for Chef Solo to find roles (default: [ascendin
g search])
  config.role_path = '/var/roles'
```

```
  # Specify the Chef log_level (default: :warn)
  config.log_level = :debug

  # Specify the path to a local JSON file with Ohai data (default: n
il)
  config.path = 'ohai.json'

  # Specify the operating platform to mock Ohai data from (default:
nil)
  config.platform = 'ubuntu'

  # Specify the operating version to mock Ohai data from (default: n
il)
  config.version = '12.04'
end
```

Values specified at the initialization of the `Runner` merge and take precedence over the global settings:

```
# Override only the operating system version (platform is still "ubu
ntu" from above)
ChefSpec::Runner.new(version: '10.04')

# Use a different operating system platform and version
ChefSpec::Runner.new(platform: 'centos', version: '5.4')

# Specify a different cookbook_path
ChefSpec::Runner.new(cookbook_path: '/var/my/other/path', role_path:
 '/var/my/roles')

# Add debug log output
ChefSpec::Runner.new(log_level: :debug).converge(described_recipe)
```

NOTE You do not *need* to specify a platform and version. However, some

cookbooks may rely on Ohai data that ChefSpec cannot not automatically generate. Specifying the `platform` and `version` keys instructs ChefSpec to load stubbed Ohai attributes from another platform using fauxhai.

Berkshelf

If you are using Berkshelf, simply require `chefspec/berkshelf` in your `spec_helper` after requiring `chefspec`:

```
# spec_helper.rb
require 'chefspec'
require 'chefspec/berkshelf'
```

Requiring this file will:

- Create a temporary working directory
- Download all the dependencies listed in your `Berksfile` into the temporary directory
- Set ChefSpec's `cookbook_path` to the temporary directory

Librarian

If you are using Librarian, simply require `chefspec/librarian` in your `spec_helper` after requiring `chefspec`:

```
# spec_helper.rb
require 'chefspec'
require 'chefspec/librarian'
```

Requiring this file will:

- Create a temporary working directory
- Download all the dependencies listed in your `Cheffile` into the temporary directory
- Set ChefSpec's `cookbook_path` to the temporary directory

NOTE In order to test the cookbook in the current working directory, you have to write your `Cheffile` like this:

```
# Cheffile
site 'http://community.opscode.com/api/v1'

cookbook 'name_of_your_cookbook', path: '.'
```

Running Specs

ChefSpec is actually a very large RSpec extension, so you can run your tests using the RSpec CLI:

```
$ rspec
```

You can also specify a specific spec to run and various RSpec command line options:

```
$ rspec spec/unit/recipes/default_spec.rb --color
```

For more information on the RSpec CLI, please see the documentation.

Making Assertions

ChefSpec asserts that resource actions have been performed. In general, ChefSpec follows the following pattern:

```
require 'chefspec'

describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new.converge(described_recipe) }

  it 'does something' do
    expect(chef_run).to ACTION_RESOURCE(NAME)
```

```
  end
end
```

where:

- *ACTION* - the action on the resource (e.g. `install`)
- *RESOURCE* - the name of the resource (e.g. `package`)
- *NAME* - the name attribute for the resource (e.g. `apache2`)

Here's a more concrete example:

```
require 'chefspec'

describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new.converge(described_recipe) }

  it 'does something' do
    expect(chef_run).to install_package('apache2')
  end
end
```

This test is asserting that the Chef run will have a *package* resource with the name *apache2* with an action of *install*.

To test that a resource action is performed with a specific set of attributes, you can call `with(ATTRIBUTES_HASH)` on the expectation, per the following example:

```
require 'chefspec'

describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new.converge(described_recipe) }

  it 'does something' do
    expect(chef_run).to modify_group('docker').with(members: ['vagra
nt'])
```

```
  end
end
```

This test is asserting that the Chef run will have a *group* resource with the name *docker*, an action of *modify*, and an attributes hash including `{ members: ['vagrant'] }`.

ChefSpec includes matchers for all of Chef's core resources using the above schema. Each resource matcher is self-documented using Yard and has a corresponding cucumber test from the examples directory.

Additionally, ChefSpec includes the following helpful matchers. They are also documented in Yard, but they are included here because they do not follow the "general pattern".

include_recipe

Assert that the Chef run included a recipe from another cookbook

```
expect(chef_run).to include_recipe('other_cookbook::recipe')
```

notify

Assert that a resource notifies another in the Chef run

```
resource = chef_run.template('/etc/foo')
expect(resource).to notify('service[apache2]').to(:restart).immediat
ely
```

subscribes

Assert that a resource subscribes to another in the Chef run

```
resource = chef_run.service('apache2')
expect(resource).to subscribe_to('template[/etc/foo]').on(:create).d
```

```
elayed
```

render_file

Assert that the Chef run renders a file (with optional content); this will match `cookbook_file`, `file`, and `template` resources and can also check the resulting content

```
expect(chef_run).to render_file('/etc/foo')
expect(chef_run).to render_file('/etc/foo').with_content('This is co
ntent')
expect(chef_run).to render_file('/etc/foo').with_content(/regex work
s too.+/)
```

Additionally, it is possible to assert which Chef phase of execution a resource is created. Given a resource that is installed at compile time using `run_action`:

```
package('apache2').run_action(:install)
```

You can assert that this package is installed during runtime using the `.at_compile_time` predicate on the resource matcher:

```
expect(chef_run).to install_package('apache2').at_compile_time
```

Similarly, you can assert that a resource is executed during convergence time:

```
expect(chef_run).to install_package('apache2').at_converge_time
```

Since "converge time" is the default behavior for all recipes, this test might be redundant and the predicate could be dropped depending on your situation.

For more complex examples, please see the examples directory or the Yard documentation.

Setting node Attributes

Node attribute can be set when creating the `Runner`. The initializer yields a block that gives full access to the node object:

```
describe 'example::default' do
  let(:chef_run) do
    ChefSpec::Runner.new do |node|
      node.set['cookbook']['attribute'] = 'hello'
    end.converge(described_recipe)
  end
end
```

Automatic attributes

ChefSpec provides mocked automatic Ohai data using fauxhai. To mock out `automatic` attributes, you must use the `automatic` key:

```
describe 'example::default' do
  let(:chef_run) do
    ChefSpec::Runner.new do |node|
      node.automatic['memory']['total'] = '512kB'
    end.converge(described_recipe)
  end
end
```

The `node` that is returned is actually a `Chef::Node` object.

To set an attribute within a specific test, set the attribute in the `it` block and then **(re-)converge the node**:

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new } # Notice we don't converge
 here
```

```
  it 'performs the action' do
    chef_run.node.set['cookbook']['attribute'] = 'hello'
    chef_run.converge(described_recipe) # The converge happens insid
e the test

    expect(chef_run).to do_something
  end
end
```

Using Chef Zero

By default, ChefSpec runs in Chef Solo mode. As of ChefSpec v3.1.0, you can ask ChefSpec to create an in-memory Chef Server during testing using ChefZero. This is especially helpful if you need to support searching or data bags.

To use the ChefSpec server, simply require the module in your `spec_helper`:

```
# spec_helper.rb
require 'chefspec'
require 'chefspec/server'
```

This will automatically create a Chef server, synchronize all the cookbooks in your `cookbook_path`, and wire all the internals of Chef together. Recipe calls to `search`, `data_bag` and `data_bag_item` will now query the ChefSpec server.

DSL

The ChefSpec server includes a collection of helpful DSL methods for populating data into the Chef Server.

Create a client:

```
ChefSpec::Server.create_client('my_client', { admin: true })
```

Create a data bag (and items):

```
ChefSpec::Server.create_data_bag('my_data_bag', {
  'item_1' => {
    'password' => 'abc123'
  },
  'item_2' => {
    'password' => 'def456'
  }
})
```

Create an environment:

```
ChefSpec::Server.create_environment('my_environment', { description:
 '...' })
```

Create a node:

```
ChefSpec::Server.create_node('my_node', { run_list: ['...'] })
```

You may also be interested in the `stub_node` macro, which will create a new `Chef::Node` object and accepts the same parameters as the Chef Runner and a Fauxhai object:

```
www = stub_node(platform: 'ubuntu', version: '12.04') do |node|
        node.set['attribute'] = 'value'
      end

# `www` is now a local Chef::Node object you can use in your test. T
o push this
# node to the server, call `create_node`:

ChefSpec::Server.create_node(www)
```

Create a role:

```
ChefSpec::Server.create_role('my_role', { default_attributes: {} })

# The role now exists on the Chef Server, you can add it to a node's
 run_list
# by adding it to the `converge` block:
let(:chef_run) { ChefSpec::Runner.new.converge(described_recipe, 'ro
le[my_role]') }
```

NOTE The ChefSpec server is empty at the start of each example to avoid interdependent tests. You can use `before` blocks to load data before each test.

Stubbing

Command

Given a recipe with shell guard:

```
template '/tmp/foo.txt' do
  not_if 'grep /tmp/foo.txt text'
end
```

ChefSpec will raise an error like:

```
Real commands are disabled. Unregistered command: `grep /tmp/foo.txt
 text`

You can stub this command with:

 stub_command("grep /tmp/foo.txt text").and_return(true)

================================================================
```

Just like the error message says, you must stub the command result. This can be

done inside a `before` block or inside the `it` block, and the stubbing method accepts both a value or Ruby code. If provided a value, the result is static. If provided a Ruby block, the block is evaluated each time the search is called.

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
    stub_command("grep /tmp/foo.txt text").and_return(true)
  end
end
```

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
    stub_command("grep /tmp/foo.txt text") { rand(50)%2 == 0 }
  end
end
```

Data Bag & Data Bag Item

NOTE This is not required if you are using a ChefSpec server.

Given a recipe that executes a `data_bag` method:

```
data_bag('users').each do |user|
  data_bag_item('users', user['id'])
end
```

ChefSpec will rails an error like:

```
Real data_bags are disabled. Unregistered data_bag: data_bag(:users)
```

```
You can stub this data_bag with:

  stub_data_bag("users").and_return([])

================================================================
```

Just like the error message says, you must stub the result of the `data_bag` call. This can be done inside a `before` block or inside the `it` block, and the stubbing method accepts both a value or Ruby code. If provided a value, the result is static. If provided a Ruby block, the block is evaluated each time the search is called.

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
    stub_data_bag('users').and_return([])
  end
end
```

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
    stub_data_bag('users').and_return(['svargo', 'francis'])

    stub_data_bag_item('users', 'svargo').and_return({ ... })
    stub_data_bag_item('users', 'francis') { (ruby code) }
  end
end
```

If you are using **Encrypted Data Bag Items**, you'll need to dive into the RSpec layer and stub that class method instead:

```
describe 'example::default' do
  before do
    Chef::EncryptedDataBagItem.stub(:load).with('users', 'svargo').a
nd_return(...)
  end
end
```

Search

NOTE This is not required if you are using a ChefSpec server.

Because ChefSpec is a unit-testing framework, it is recommended that all third-party API calls be mocked or stubbed. ChefSpec exposes a helpful RSpec macro for stubbing search results in your tests. If you converge a Chef recipe that implements a `search` call, ChefSpec will throw an error like:

```
Real searches are disabled. Unregistered search: search(:node, 'name
:hello')

You can stub this search with:

  stub_search(:node, 'name:hello') {  }

==========================================================
```

Just like the error message says, you must stub the search result. This can be done inside a `before` block or inside the `it` block, and the stubbing method accepts both a value or Ruby code. If provided a value, the result is static. If provided a Ruby block, the block is evaluated each time the search is called.

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
```

```
    stub_search(:node, 'name:hello').and_return([])
  end
end
```

```
describe 'example::default' do
  let(:chef_run) { ChefSpec::Runner.new }

  before do
    stub_search(:node, 'name:hello') { (ruby_code) }
  end
end
```

Reporting

ChefSpec can generate a report of resources read over resources tested. Please note, this feature is currently in beta phases and may not be 100% accurate. That being said, it is currently the only code coverage tool available for Chef recipes.

To generate the coverage report, add the following to your `spec_helper.rb` before you require any "Chef" code:

```
require 'chefspec'
ChefSpec::Coverage.start!

# Existing spec_helper contents...
```

By default, that method will output helpful information to standard out:

```
ChefSpec Coverage report generated at '.coverage/results.json':

  Total Resources:   6
  Touched Resources: 1
  Touch Coverage:    16.67%
```

```
Untouched Resources:

  package[git]                bacon/recipes/default.rb:2
  package[build-essential]    bacon/recipes/default.rb:3
  package[apache2]            bacon/recipes/default.rb:4
  package[libvrt]             bacon/recipes/default.rb:5
  package[core]               bacon/recipes/default.rb:6
```

By default, ChefSpec will test all cookbooks that are loaded as part of the Chef Client run. If you have a cookbook with many dependencies, this may be less than desireable. To restrict coverage reporting against certain cookbooks, `ChefSpec::Coverage` yields a block:

```
ChefSpec::Coverage.start! do
  add_filter 'vendor/cookbooks'
end
```

The `add_filter` method accepts a variety of objects. For example:

```
ChefSpec::Coverage.start! do
  # Strings are interpreted as file paths, with a forward anchor
  add_filter 'vendor/cookbooks'

  # Regular expressions must be escaped, but provide a nicer API for
 negative
  # back tracking
  add_filter /cookbooks\/(?!omnibus)/

  # Custom block filters yield a {Chef::Resource} object - if the bl
ock
  # evaluates to true, it will be filtered
  add_filter do |resource|
    # Bob's cookbook's are completely untested! Ignore them until he
 gets his
```

```
    # shit together.
    resource.source_file =~ /cookbooks\/bob-(.+)/
  end
end
```

For more complex scenarios, you can create a custom `Filter` object that inherits from `ChefSpec::Coverage::Filter` and implements the `matches?` method.

```
class CustomFilter < ChefSpec::Coverage::Filter
  def initialize(arg1, arg2, &block)
    # Create a custom initialization method, do some magic, etc.
  end

  def matches?(resource)
    # Custom matching logic in here - anything that evaluates to "tr
ue" will be
    # filtered.
  end
end

ChefSpec::Converage.start! do
  add_filter CustomFilter.new('foo', :bar)
end
```

If you are using ChefSpec's Berkshelf plugin, a filter is automatically created for you. If you would like to ignore that filter, you can `clear` all the filters before defining your own:

```
ChefSpec::Coverage.start! do
  filters.clear

  # Add your custom filters now
end
```

Mocking Out Environments

If you want to mock out `node.chef_environment`, you'll need to use RSpec mocks/stubs twice:

```
let(:chef_run) do
  ChefSpec::Runner.new do |node|
    # Create a new environment (you could also use a different :let block or :before block)
    env = Chef::Environment.new
    env.name 'staging'

    # Stub the node to return this environment
    node.stub(:chef_environment).and_return(env.name)

    # Stub any calls to Environment.load to return this environment
    Chef::Environment.stub(:load).and_return(env)
  end.converge('cookbook::recipe')
end
```

There is probably a better/easier way to do this. If you have a better solution, please open an issue or Pull Request so we can make this less painful :)

Testing LWRPs

WARNING Cookbooks with dashes (hyphens) are difficult to test with ChefSpec because of how Chef classifies objects. We recommend naming cookbooks with underscores (_) instead of dashes (-).

ChefSpec overrides all providers to take no action (otherwise it would actually converge your system). This means that the steps inside your LWRP are not actually executed. If an LWRP performs actions, those actions are never executed or added to the resource collection.

In order to run the actions exposed by your LWRP, you have to explicitly tell the

`Runner` to step into it:

```
require 'chefspec'

describe 'foo::default' do
  let(:chef_run) { ChefSpec::Runner.new(step_into: ['my_lwrp']).converge('foo::default') }

  it 'installs the foo package through my_lwrp' do
    expect(chef_run).to install_package('foo')
  end
end
```

NOTE: If your cookbook exposes LWRPs, it is highly recommended you also create a `libraries/matchers.rb` file as outlined below in the "Packaging Custom Matchers" section. **You should never `step_into` an LWRP unless you are testing it. Never `step_into` an LWRP from another cookbook!**

Packaging Custom Matchers

ChefSpec exposes the ability for cookbook authors to package custom matchers inside a cookbook so that other developers may take advantage of them in testing. This is done by creating a special library file in the cookbook named `matchers.rb`:

```
# cookbook/libraries/matchers.rb

if defined?(ChefSpec)
  def my_custom_matcher(resource_name)
    ChefSpec::Matchers::ResourceMatcher.new(resource, action, resource_name)
  end
end
```

1. The entire contents of this file must be wrapped with the conditional clause checking if `ChefSpec` is defined.
2. Each matcher is actually a top-level method. The above example corresponds to the following RSpec test:

```
expect(chef_run).to my_custom_matcher('...')
```

3. `ChefSpec::Matchers::ResourceMatcher` accepts three parameters:

    1. The name of the resource to find in the resource collection (i.e. the name of the LWRP).
    2. The action that resource should receive.
    3. The value of the name attribute of the resource to find. (This is typically proxied as the value from the matcher definition.)

ChefSpec's built-in `ResourceMatcher` *should* satisfy most common use cases for packaging a custom matcher with your LWRPs. However, if your cookbook is extending Chef core or is outside of the scope of a traditional "resource", you may need to create a custom matcher. For more information on custom matchers in RSpec, please watch the Railscast on Custom Matchers or look at some of the other custom matchers in ChefSpec's source code.

Example

Suppose I have a cookbook named "motd" with a resource/provider "message".

```
# motd/resources/message.rb
actions :write
default_action :write

attribute :message, name_attribute: true
```

```
# motd/providers/message.rb
action :write do
  # ...
```

```
end
```

Chef will dynamically build the `motd_message` LWRP at runtime that can be used in the recipe DSL:

```
motd_message 'my message'
```

You can package a custom ChefSpec matcher with the motd cookbook by including the following code in `libraries/matchers.rb`:

```
# motd/libraries/matchers.rb
if defined?(ChefSpec)
  def write_motd_message(message)
    ChefSpec::Matchers::ResourceMatcher.new(:motd_message, :write, m
essage)
  end
end
```

Other developers can write RSpec tests against your LWRP in their cookbooks:

```
expect(chef_run).to write_motd_message('my message')
```

Don't forget to include documentation in your cookbook's README noting the custom matcher and it's API!

Writing Custom Matchers

If you are testing a cookbook that does not package it's LWRP matchers, you can create your own following the same pattern as the "Packaging Custom Matchers" section. Simply, create a file at `spec/support/matchers.rb` and add your resource matchers:

```
# spec/support/matchers.rb
```

```
def my_custom_matcher(resource_name)
  ChefSpec::Matchers::ResourceMatcher.new(:resource, :action, resour
ce_name)
end
```

Then require this file in your `spec_helper.rb` so the matcher can be used:

```
require_relative 'support/matchers'
```

Please use this as a *temporary* solution. Consider sending a Pull Request to the LWRP author(s) packaging the custom resource matchers (see previous section).

Expecting Exceptions

In Chef 11, custom formatters were introduced and ChefSpec uses a custom formatter to suppress Chef Client output. In the event of a convergence failure, ChefSpec will output the error message from the run to help you debug:

```
================================================================================
============
Recipe Compile Error in apt_package/recipes/install.rb
================================================================================
============

RuntimeError
------------
RuntimeError

Cookbook Trace:
---------------
  .../apt_package/recipes/install.rb:1:in `from_file'
  .../apt_package/spec/install_spec.rb:4:in `block (2 levels) in <to
p (required)>'
  .../apt_package/spec/install_spec.rb:7:in `block (2 levels) in <to
```

```
p (required)>'

Relevant File Content:
----------------------
.../apt_package/recipes/install.rb:

  1>> raise RuntimeError
  2:
  3:  apt_package 'default_action'
```

This output is automatically silenced when using RSpec's `raise_error` matcher:

```
let(:chef_run) { ChefSpec::Runner.new.converge('cookbook::recipe') }

it 'raises an error' do
  expect {
    chef_run
  }.to raise_error
end
```

You can also assert that a particular error was raised. If the error matches the given type, the output is suppressed. If not, the test fails and the entire stack trace is presented.

```
let(:chef_run) { ChefSpec::Runner.new.converge('cookbook::recipe') }

it 'raises an error' do
  expect {
    chef_run
  }.to raise_error(RuntimeError)
end
```

Testing Roles

Even though ChefSpec is cookbook-centric, you can still converge multiple recipes and roles in a single `ChefSpec::Runner` instance. Given a cookbook "bacon" with a default recipe:

```
# cookbooks/bacon/recipes/default.rb
package 'foo'
```

and a default attributes file:

```
# cookbooks/bacon/attributes/default.rb
default['bacon']['temperature'] = 200
```

and a role "breakfast":

```
# roles/breakfast.rb
default_attributes(
  'bacon' => {
    'temperature' => 150 # NOTE: This is different from the default
value
  }
)
run_list([
  'recipe[bacon::default]'
])
```

You can test that the role is appropriately applied by telling the `ChefSpec::Runner` to converge on the *role* instead of a recipe:

```
let(:chef_run) { ChefSpec::Runner.new.converge('role[breakfast]') }
```

Assert that the run_list is properly expanded:

```
expect(chef_run).to include_recipe('bacon::default')
```

Assert that the correct attribute is used:

```
expect(chef_run.node['bacon']['temperature']).to eq(150)
```

NOTE If your roles live somewhere outside of the expected path, you must set `RSpec.config.role_path` to point to the directory containing your roles **before** invoking the `#converge` method!

```
RSpec.configure do |config|
  config.role_path = '/var/my/roles' # global setting
end

# - OR -

ChefSpec::Runner.new(role_path: '/var/my/roles') # local setting
```

Faster Specs

ChefSpec aims to provide the easiest and simplest path for new users to write RSpec examples for Chef cookbooks. In doing so, it makes some sacrifices in terms of speed and agility of execution. In other words, ChefSpec favors "speed to develop" over "speed to execute". Many of these decisions are directly related to the way Chef dynamically loads resources at runtime.

If you understand how RSpec works and would like to see some significant speed improvements in your specs, you can use the `ChefSpec::Cacher` module inspired by Juri Timošin. Just require the cacher module in your spec helper.

```
# spec_helper.rb
require 'chefspec/cacher'
```

Next, convert all your `let` blocks to `cached`:

```
# before
```

```
let(:chef_run) { ChefSpec::Runer.new }

# after
cached(:chef_run) { ChefSpec::Runner.new }
```

Everything else should work the same. Be advised, as the method name suggests, this will cache the results of your Chef Client Run for the **entire RSpec example**. This makes stubbing more of a challenge, since the node is already converged. For more information, please see Juri Timošin's blog post on faster specs as well as the discussion in #275.

Media & Third-party Tutorials

- CustomInk's Testing Chef Cookbooks
- Jake Vanderdray's Practical ChefSpec
- Jim Hopp's excellent Test Driven Development for Chef Practitioners
- Joshua Timberman's Starting ChefSpec Examples
- Juri Timošin's post on faster specs
- Seth Vargo's Chef recipe code coverage
- Seth Vargo's TDDing tmux talk
- Stephen Nelson Smith's Test-Driven Infrastructure with Chef

Development

1. Fork the repository from GitHub.
2. Clone your fork to your local machine:

```
$ git clone git@github.com:USER/chefspec.git
```

3. Create a git branch

```
$ git checkout -b my_bug_fix
```

4. **Write tests**

5. Make your changes/patches/fixes, committing appropriately
6. Run the tests: `bundle exec rake`
7. Push your changes to GitHub
8. Open a Pull Request

ChefSpec is on Travis CI which tests against multiple Chef and Ruby versions.

If you are contributing, please see the Contributing Guidelines for more information.

License

MIT - see the accompanying LICENSE file for details.