Testing Resources in Recipes

The default recipe we refactored moved the resources into individual recipes that will promote their ability to be composed in other cookbooks. Now its time to take a look at the resources we defined and explore writing examples to verify their state as well.

Objectives

After completing this module, you should be able to:

- Test resources within a recipe using ChefSpec

In this module you will learn how to test resources within a recipe using ChefSpec.

EXERCISE

Testing Remaining Resources

No resources left behind!

Objective:

- ❑ Write and execute tests for the Install recipe
- ❑ Verify the test validates the recipe

If we continued to use the mutation testing approach we would find similar problems with in the other recipes that we developed. Together let's work through defining examples for this recipe and then you will have a lab later to complete the remaining recipes.

Generated Recipes Also Generate Specs

```
> tree spec
```

```
spec
├── spec_helper.rb
└── unit
    └── recipes
        ├── configuration_spec.rb
        ├── default_spec.rb
        ├── install_spec.rb
        └── service_spec.rb
```


Back when we generated the recipe with the 'chef' command-line utility a matching specification file was also generated. Similar to the default recipe specification the install recipe specification contains a single example that ensures that the chef run completes without error.

Execute the Install Specification

```
> chef exec rspec spec/unit/recipes/install_spec.rb
```

```
.

Finished in 0.46874 seconds (files took 4.24 seconds to load)
1 example, 0 failures
```


Using 'rspec' we can verify that the one example completes successfully.

Update the ChefSpec Platform

~/apache/spec/unit/recipes/install_spec.rb

```
require 'spec_helper'

describe 'apache::install' do
  context 'When all attributes are default, on an CentOS 6.7' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.7')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end
  end
end
```


Again it is important that your tests describe the current system that this recipe is working on. Especially if you build your cookbook to support multiple platforms.

Add a Pending Test to Verify the Package

~/apache/spec/unit/recipes/install_spec.rb

```
    # ... START OF THE SPEC FILE ...

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    it 'installs the necessary package'
  end
end
```


The install recipe installs the necessary the necessary software for the webserver. We can start by writing a pending example.

REFERENCE

ChefSpec Documentation

Find within the documentation examples of testing packages

https://github.com/sethvargo/chefspec/tree/master/examples/package

Now it is time returned to the documentation. Again, the ChefSpec documentation contains a lot of examples in the README and the examples directory. Using either of those find an example of an expectation expressing that a packaged is installed.

Write the Test to Verify the Package

~/apache/spec/unit/recipes/install_spec.rb

```
    # ... START OF THE SPEC FILE ...

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    it 'installs the necessary package' do
      expect(chef_run).to install_package('httpd')
    end
  end
end
```


With a good example we found in the documentation we can return to the example and define the example. In our case we want to assert that the the chef run installs the package 'httpd'.

Write the Test to Verify the Package

~/apache/spec/unit/recipes/install_spec.rb

```
# ... START OF THE SPEC FILE ...

it 'converges successfully' do
  expect { chef_run }.to_not raise_error
end

it 'installs the necessary package' do
  expect(chef_run).to install_package('httpd')
end
end
end
```

resource's action

resource

resource's name

Expressing an expectation for the state of resources in the 'Resource Collection' uses a particular matcher. Express the name of the action joined together with the type of the resource and has the parameter that is the name of the resource.

The expectation defined here is slightly different than the previous example. In the first example the expect uses braces. This is Ruby's shorthand notation to represent a block. The reason in this expectation we want to use a block is that if the chef run were to raise an error we need to catch it. Catching it requires that we wrap the code we want to execute within a block.

Using the parenthesis is passing the 'chef_run' helper as a parameter to the 'expect' method. In this instance we do not expect an error to take place and instead want to make assertions on the state of the chef run. If an error were to be raised the expectation would not catch it and instead of the expectation failing you would see an error message.

Execute the Test to See it Pass

```
> chef exec rspec spec/unit/recipes/install_spec.rb
```

```
..

Finished in 0.73662 seconds (files took 4.4 seconds to load)
2 examples, 0 failures
```


When we are done writing this expectation and execute the test suite we see that we now have 2 examples that both pass.

EXERCISE

Testing Remaining Resources

No resources left behind!

Objective:

- ✓ Write and execute tests for the Install recipe
- ❑ Verify the test validates the recipe

We now have an expectation that expresses the state for the install recipe. But before we declare victory it is time to verify that the expectations truly are working.

PROBLEM

It's Quiet. Too Quiet.

When a test passes immediately without having to write code (or if the code has already been written) it is time to be concerned. This is one of those moments we should ensure that the tests are working by mutating that code.

If a test passes and you have never seen it fail. How do you know it works? Without ever seeing a failure there is situation where we could be seeing a 'false positive'. This is because we did not develop this expectation with the test first. In this instance we have not done anything wrong. We simply need to ensure that the expectation we define will fail if we were to modify the code that we are testing.

To do that it is time for us to return to the recipe and modify it, mutate it, to ensure that the test fails.

Comment Out the Resource

~/apache/recipes/install.rb

```
#
# Cookbook Name:: apache
# Recipe:: install
#
# Copyright (c) 2017 The Authors, All Rights Reserved.
# package 'httpd'
```


One simple mutation is to remove the resource by commenting it out or removing it. We could also choose to rename the name of the resource.

Execute the Test to See it Fail

```
> chef exec rspec spec/unit/recipes/install_spec.rb
```

```
.F

Failures:

  1) apache::install When all attributes are default, on an CentOS
6.7 installs the appropriate package
     Failure/Error: expect(chef_run).to install_package('httpd')
       expected "package[httpd]" with action :install to be in
Chef run. Other package resources:
```


Returning to run the tests we now see that there is an error in the execution. The change that we made to the recipe, the removal of the resource, generates this error. We can state with confidence that the expectation that we defined properly insures our expectations about the 'Resource Collection'.

Uncomment Out the Resource

~/apache/recipes/install.rb

```
#
# Cookbook Name:: apache
# Recipe:: install
#
# Copyright (c) 2017 The Authors, All Rights Reserved.
package 'httpd'
```


Time to restore the mutation we introduced.

Execute the Test to See it Pass

```
> chef exec rspec spec/unit/recipes/install_spec.rb
```

```
..

Finished in 0.73662 seconds (files took 4.4 seconds to load)
2 examples, 0 failures
```


Verify that all the examples complete successfully.

EXERCISE

Testing Remaining Resources

No resources left behind!

Objective:

- ✓ Write and execute tests for the Install recipe
- ✓ Verify the test validates the recipe

We walked through ensuring this recipe has the necessary expectations defined.

LAB

Test the Remaining Recipes

- ❑ Write a pending example
- ❑ Find the ChefSpec implementation
- ❑ Verify that the new example passes
- ❑ Mutate the recipe to generate a failure
- ❑ Restore the code in the recipe
- ❑ Verify that all examples pass

❖ **Repeat this series of steps for the configuration recipe and service recipe**

Now it is your turn. Using the same strategy it is time to address the remaining recipes within the cookbook.

Instructor Note: Allow 15 minutes to complete this exercise

Update the ChefSpec Platform

~/apache/spec/unit/recipes/service_spec.rb

```
require 'spec_helper'

describe 'apache::install' do
  context 'When all attributes are default, on an CentOS 6.7' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.7')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end
  end
end
```

Write the Tests to Verify the Service

~/apache/spec/unit/recipes/service_spec.rb

```
# ... START OF THE SPEC FILE ...

    it 'starts the necessary service' do
      expect(chef_run).to start_service('httpd')
    end

    it 'enables the necessary service' do
      expect(chef_run).to enable_service('httpd')
    end
  end
end
```


Let's review the final resulting specification for only the service recipe. We defined two examples. One that states the expectation that the necessary service has been started. The other states the expectation that the necessary service has been enabled.

Instructor Note: We are showing the final concluding content and not the workflow.

Execute the Tests to See it Pass

```
> chef exec rspec spec/unit/recipes/service_spec.rb
```

```
...

Finished in 0.93685 seconds (files took 4.28 seconds to load)
3 examples, 0 failures
```


Verifying the examples we see three passing examples.

LAB

Test the Remaining Recipes

- ✓ Write a pending example
- ✓ Find the ChefSpec implementation
- ✓ Verify that the new example passes
- ✓ Mutate the recipe to generate a failure
- ✓ Restore the code in the recipe
- ✓ Verify that all examples pass

❖ **Repeat this series of steps for the configuration recipe and service recipe**

Congratulations. Now you have completed writing unit tests for the remaining resources across all our recipes.

Instructor Note: We did not review the configuration recipe.

CONCEPT

rspec

When you run **`rspec`** without any paths it will automatically find and execute all the *"_spec.rb"* files within the '*spec*' directory.

Running 'rspec' as we have during this and the last section has shown that we can provide a file and it will evaluate the examples within that file. Now that we have examples spread across multiple recipes it would be nice to be able to run them all at once. And actually that is how RSpec is designed to work by default. When you run 'rspec' with no paths it will automatically find all specification files defined in the 'spec' directory.

It is important to note that all specification files must end with an '_spec.rb' for them to found by RSpec.

Execute All the Tests in the Spec Directory

```
> chef exec rspec
```

```
........

Finished in 2.08 seconds (files took 4.29 seconds to load)
8 examples, 0 failures
```


Let's verify every example across all the recipe specification files. In this output we see 'rspec' found 8 examples found all of them passing all within 4.29 seconds.

The execution time of RSpec varies based on the specifications, the version of ChefSpec, the power of the workstation, and the platform.

Let's have a discussion.

Instructor Note: This output was generated on a Amazon Web Services t1.micro running CentOS 6.7 installed with Chef DK 0.11.0.

DISCUSSION

Discussion

What value does it bring to validate that the resources take the appropriate action?

Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

DISCUSSION

Q&A

What questions can we answer for you?

CHEF

Before we complete this section, let us pause for questions.

Morning

Introduction
Why Write Tests? Why is that Hard?
Writing a Test First
Refactoring Cookbooks with Tests

Afternoon

Faster Feedback with Unit Testing
Testing Resources in Recipes
Refactoring to Attributes
Refactoring to Multiple Platforms

All of the resources within our recipes have expectations. Now it is time to see the value of the examples that we have defined by returning to the recipes we wrote and introduce a new requirement: using node attributes.

CHEF™