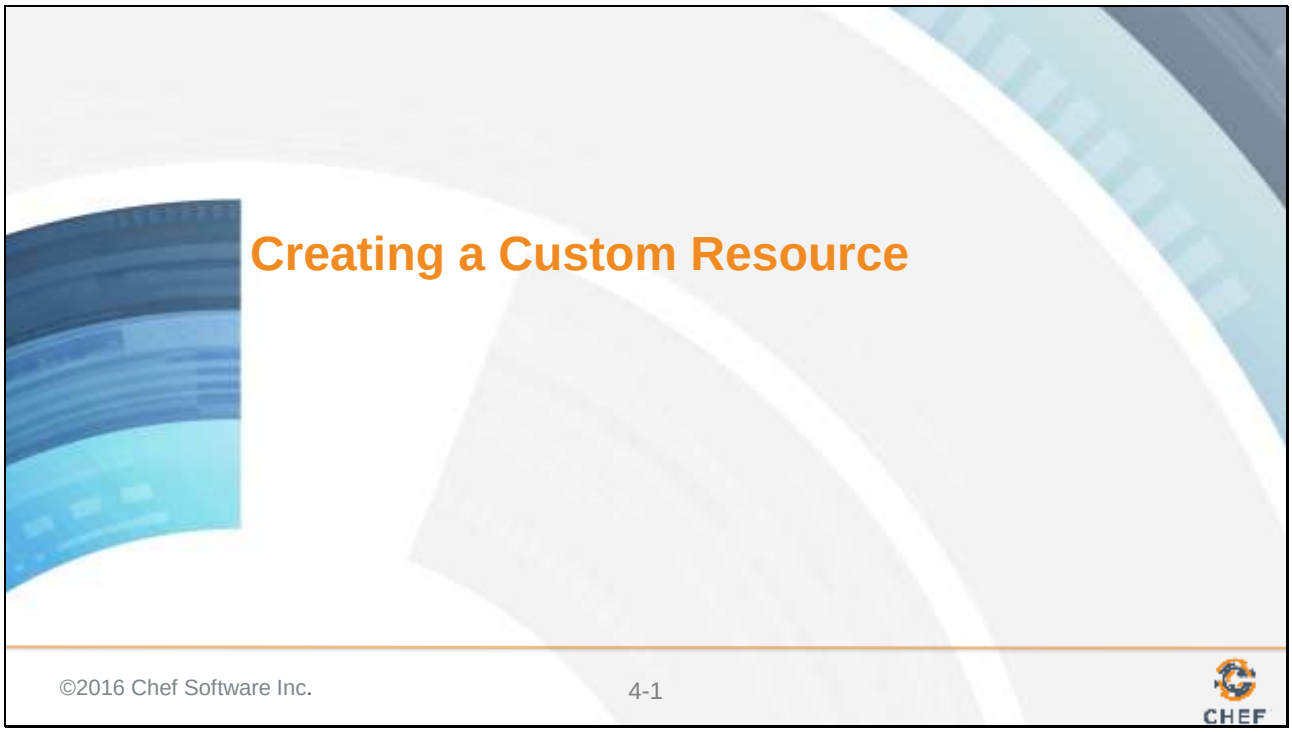
4: Creating a Custom Resource



Slide 2

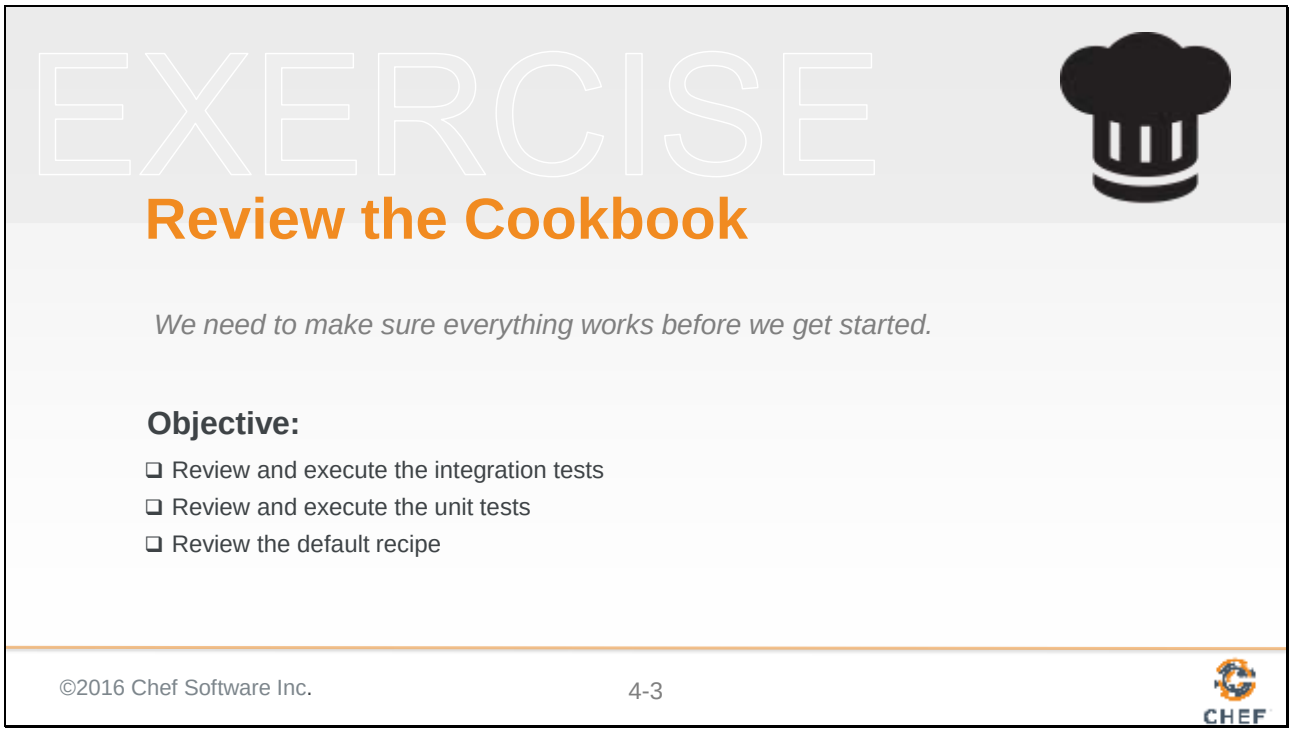
Objectives

After completing this module, you should be able to:

- Create a custom resource file
- Define a custom resource action
- Extract Chef resources into a custom resource action implementation
- Create custom resource properties

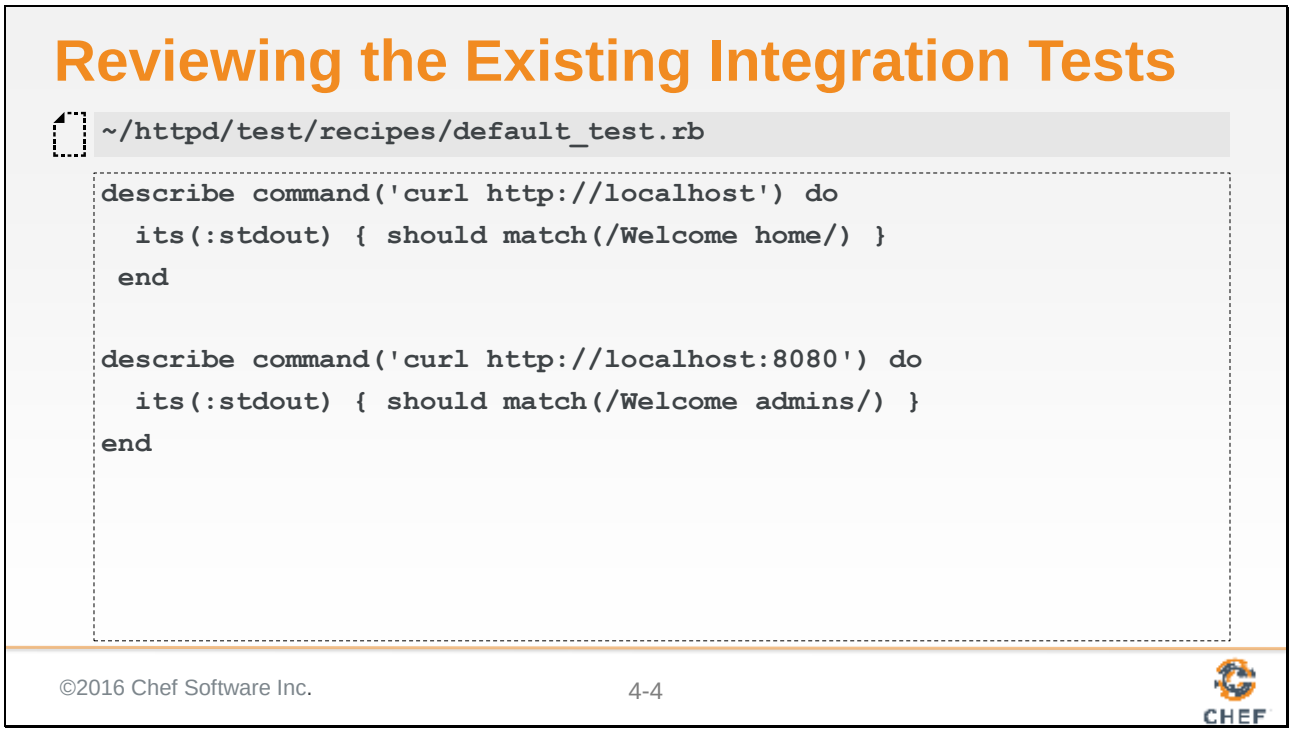
After completing this module you should be able to: create a custom resource file; define a custom resource action; and extract Chef resources into a custom resource action implementation.

Slide 3



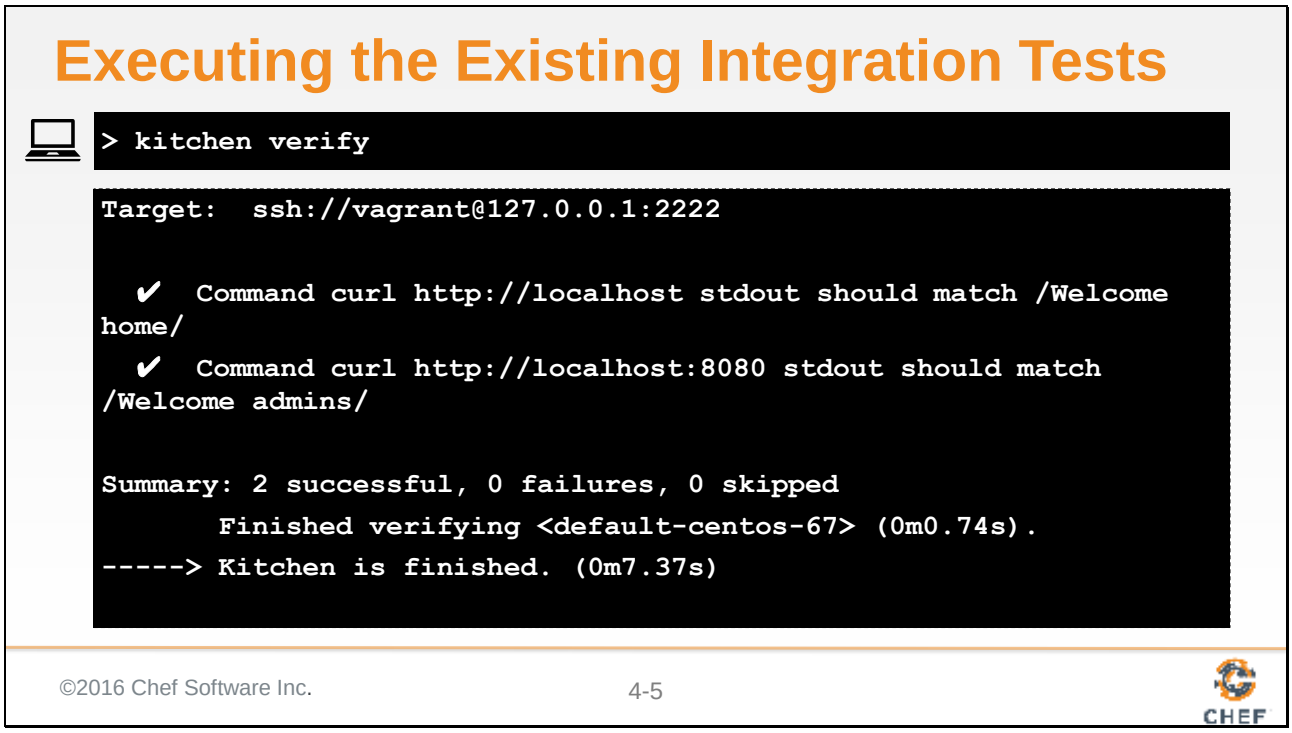
Before we begin creating this custom resource it is important to review the cookbook. We will start looking at the integration tests defined.

Slide 4



There are two tests define that assert that the a default website is available on port 80 and a second website available on port 8080. Each of these websites cater to the different possible roles one could have with the website. The standard user visits the sit on port 80 where admins visit the site on port 8080.

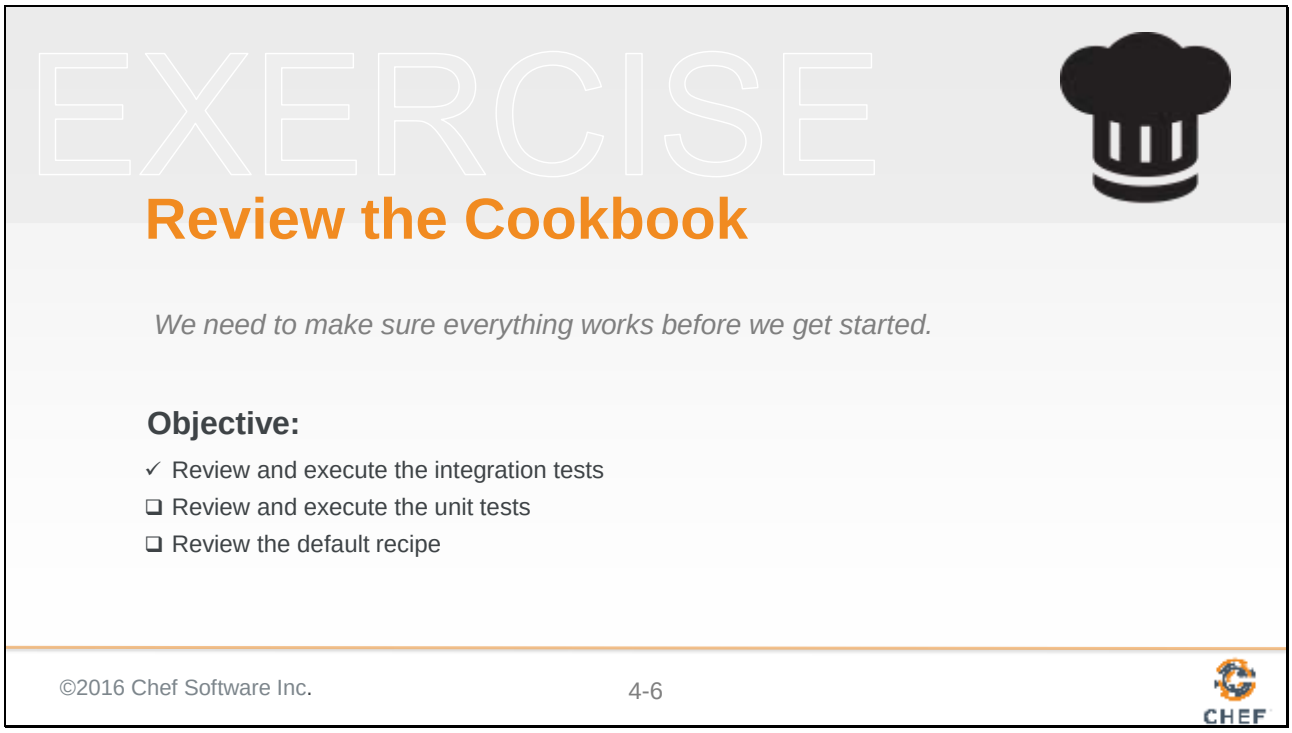
Slide 5



Before refactoring the cookbook it is important that you verify that the cookbook is in a known good state. To do that you would want to use Test Kitchen to execute the two test are defined.

Each example should pass without failure.

Slide 6



Let's examine the unit tests that are defined within the cookbook.

Slide 7

Reviewing the Existing Unit Tests

~/httpd/spec/unit/recipes/default_spec.rb

```
require 'spec_helper'

describe 'httpd::default' do
  context 'When all attributes are default, on an unspecified platform' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new
      runner.converge(described_recipe)
    end

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

# ... CONTINUES ON THE NEXT SLIDE ...
```


CHEF

There is a single specification file defined for the default recipe. The first expectation defined is the generic one that assures us that the chef run should converge without raising an error.

Slide 8

Reviewing the Existing Unit Tests

~/httpd/spec/unit/recipes/default_spec.rb

```
# ... CONTINUES FROM THE PREVIOUS SLIDE ...

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

    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
# ... CONTINUES ON THE NEXT SLIDE ...
```


The next few expectations ensure that the necessary packages are installed and the services are started and enabled.

Slide 9

Reviewing the Existing Unit Tests

~/httpd/spec/unit/recipes/default_spec.rb

```
# ... CONTINUES FROM THE PREVIOUS SLIDE ...

  describe 'for the default site' do
    it 'writes out a new home page' do
      expect(chef_run).to render_file('/var/www/html/index.html').with_content('<h1>Welcome
home!</h1>')
    end
  end

# ... CONTINUES ON THE NEXT SLIDE ...
```


The next expectation ensures that the default site has an html page that is written out and contains a small amount of content that we assume should be present within that file to ensure our guests are welcome to the site.

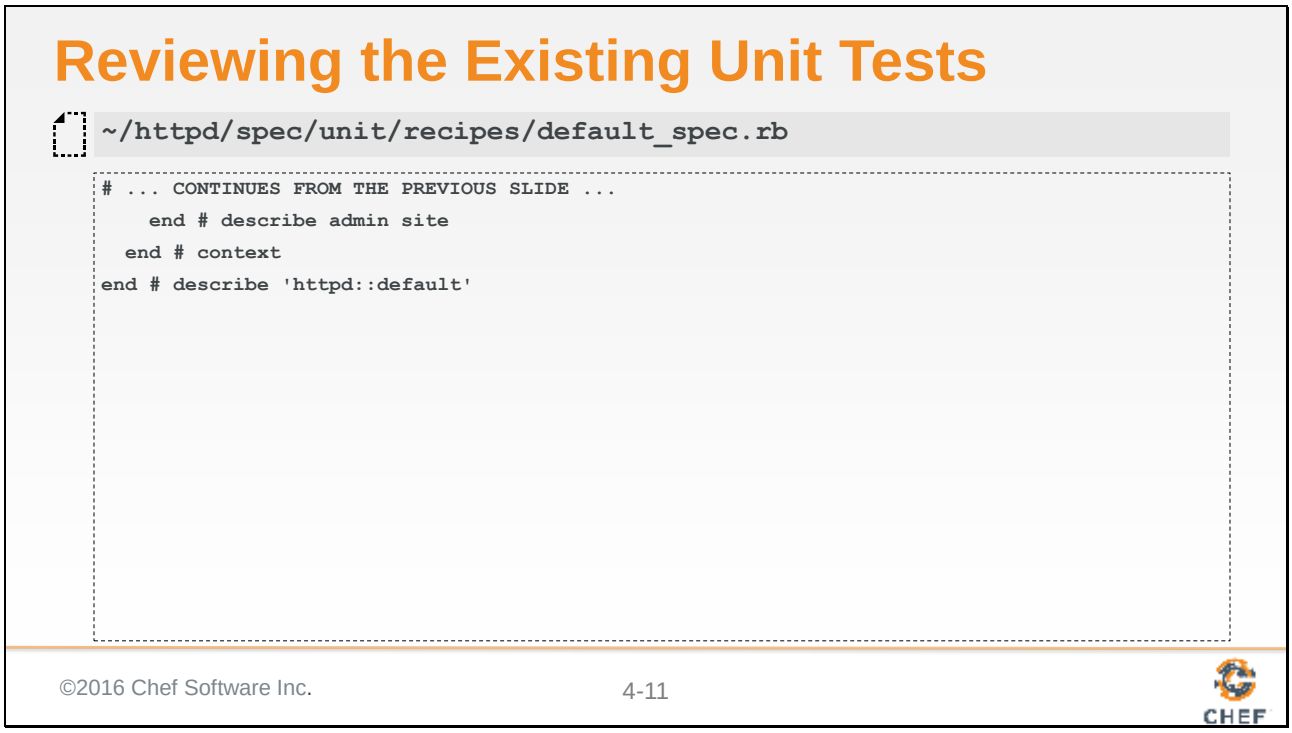
Slide 10

Reviewing the Existing Unit Tests

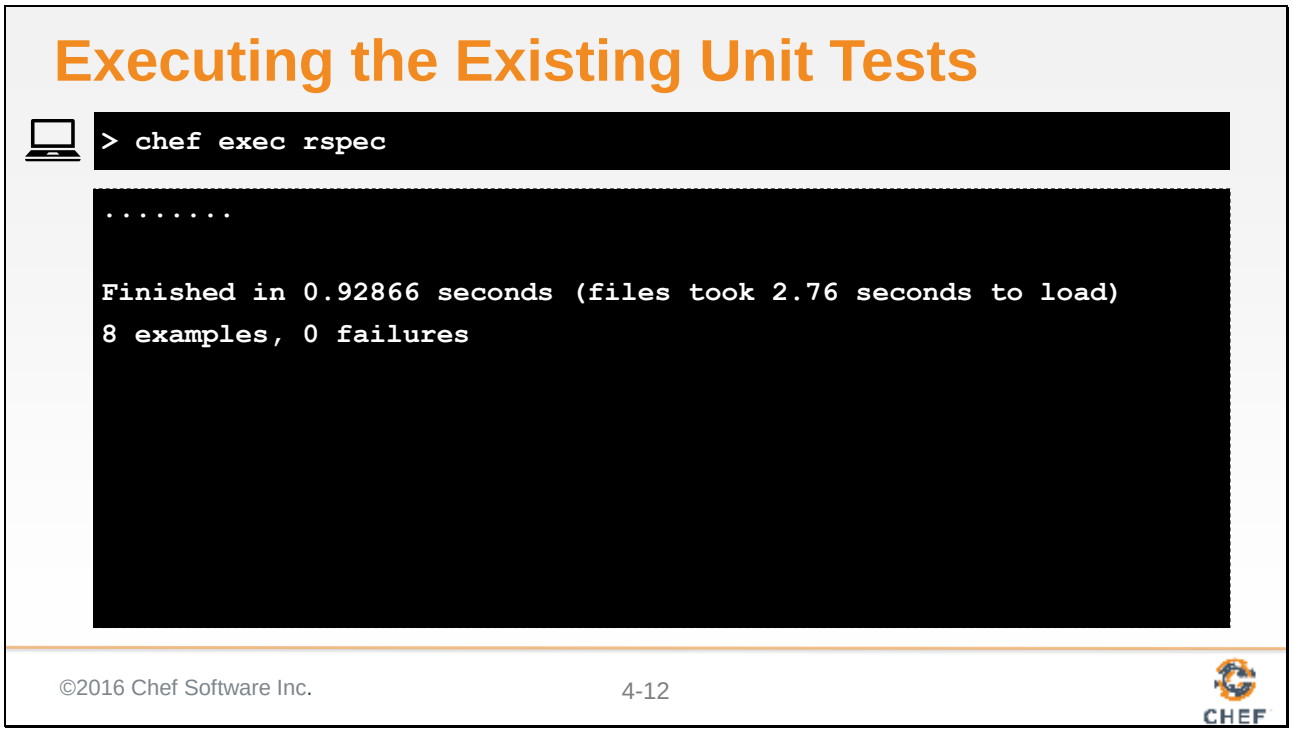
~/httpd/spec/unit/recipes/default_spec.rb

```
# ... CONTINUES FROM THE PREVIOUS SLIDE ...
  describe 'for the admin site' do
    it 'creates the directory' do
      expect(chef_run).to create_directory('/srv/apache/admins/html')
    end

    it 'creates the configuration' do
      expect(chef_run).to render_file('/etc/httpd/conf.d/admins.conf')
    end

    it 'creates a new home page' do
      expect(chef_run).to render_file('/srv/apache/admins/html/index.html').wit ('<h1...h1>')
    end
# ... CONTINUES ON THE NEXT SLIDE ...
```


For the admin site we ensure that the site directory is created, a configuration file is written, and that the home page displays a welcoming message to the admins visiting the site.



This is the end of the file showing the remaining 'end' keywords necessary to properly close the blocks that were opened (with the do keyword) above. Comments follow each one to show their matching 'do' in the file above.



After reviewing the expectations it is important to execute them to ensure that all of them pass.

Slide 13

EXERCISE

Review the Cookbook

We need to make sure everything works before we get started.

Objective:

- ✓ Review and execute the integration tests
- ✓ Review and execute the unit tests
- ❑ Review the default recipe

CHEF

Finally it is time to review the default recipe.

Slide 14

Reviewing the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome home!</h1>'
end

# ... CONTINUES ON THE NEXT SLIDE ...
```

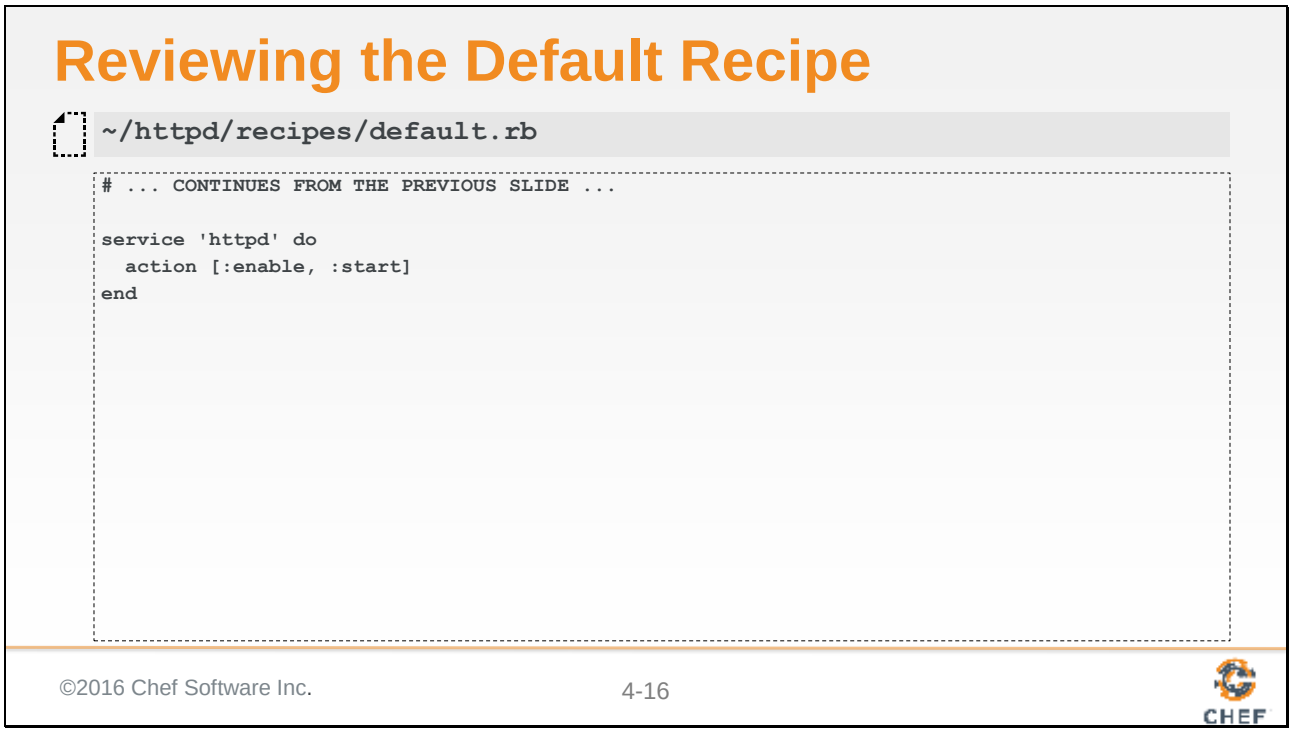

First we see that the recipe installs the necessary packages to install the web server. An html page is written out for the default site to contain the appropriate welcome message.

Reviewing the Default Recipe

~/httpd/recipes/default.rb

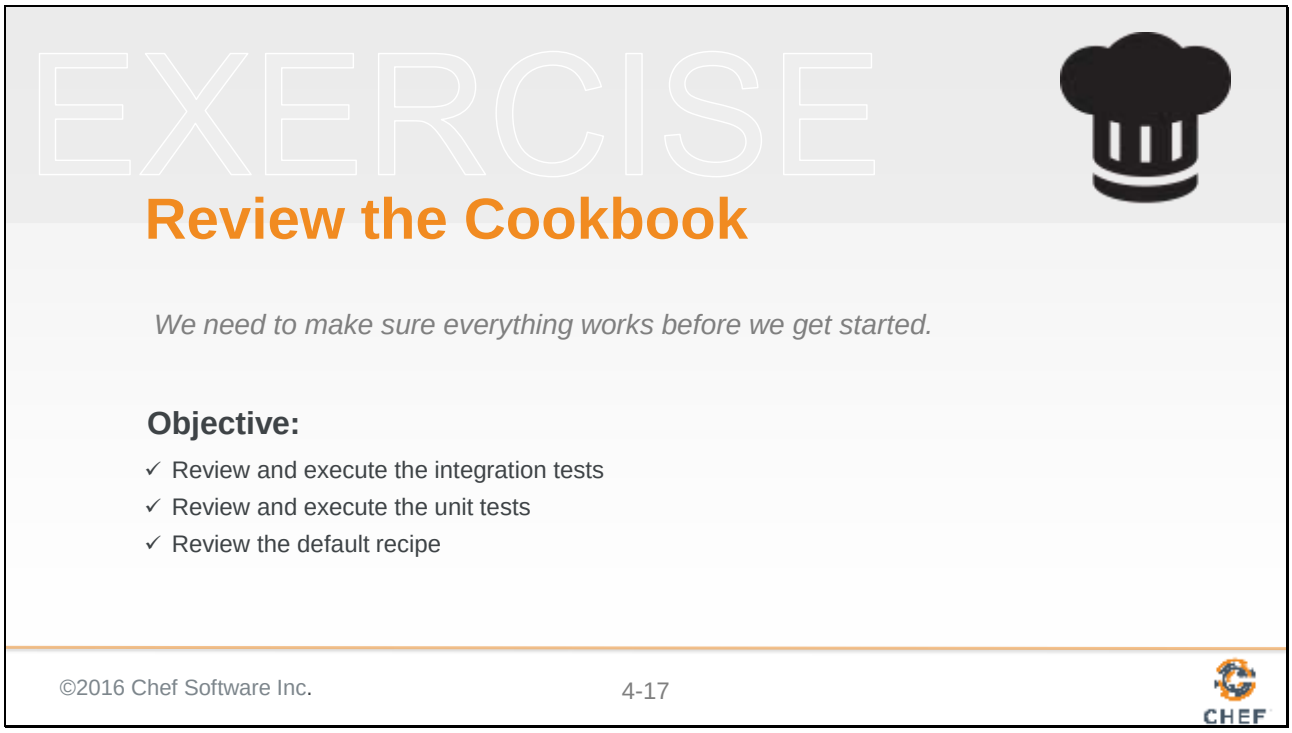
```
# ... CONTINUES FROM THE PREVIOUS SLIDE ...
directory '/srv/apache/admins/html' do
  recursive true
  mode '0755'
end

template '/etc/httpd/conf.d/admins.conf' do
  source 'conf.erb'
  mode '0644'
  variables(document_root:'/srv/apache/admins/html', port: 8080)
  notifies :restart, 'service[httpd]'
end

file '/srv/apache/admins/html/index.html' do
  content '<h1>Welcome admins!</h1>'
end
# ... CONTINUES ON THE NEXT SLIDE ...
```


The next three resources setup the admin site. First creating the directory for the admin site to store the html it will display. A configuration file is written to ensure the webserver will find the new site that we have defined. Last an index html file is added to the admin site with a welcoming message.



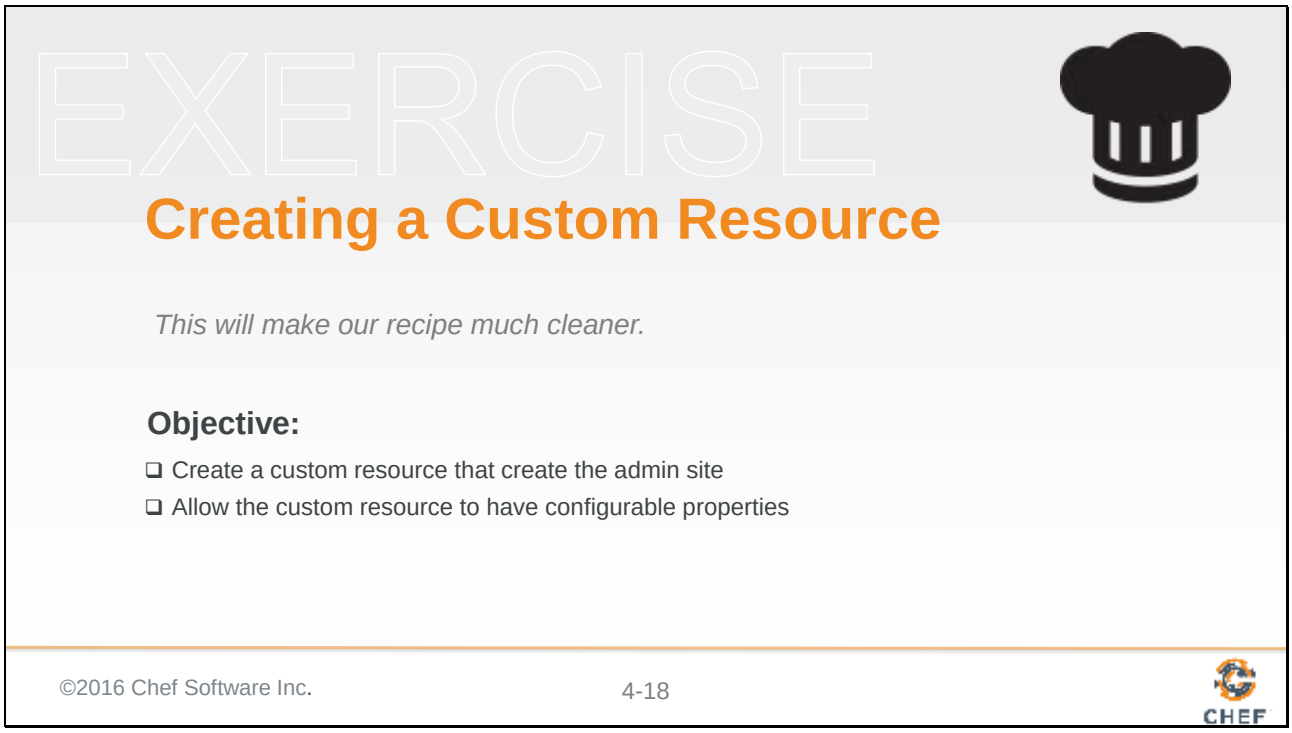
For the webserver to work correctly with the default site and the admin site the service needs to be started. We also enable the service to ensure the web server will start again if the instance this is being executed on happens to reboot.

Slide 17



Reviewing the integration tests, unit tests, and recipe gives us a good understanding of what this cookbook accomplishes.

Slide 18



With a working cookbook it is time to refactor it to use custom resources. A custom resource will help make the recipe we define express our intentions more clearly and allow us to hide some of the implementation details that make it harder for us to at-a-glance understand what a recipe is accomplishing. It will also assist us if we wanted to support multiple different sites for other roles that have yet been defined.

Generating a Custom Resource

```
> chef generate lwrp vhost
```

```
Compiling Cookbooks...
Recipe: code_generator::lwrp
  * directory[/Users/franklinwebber/scratch/extending-cookbook/httpd/resources] action create
    - create new directory /Users/franklinwebber/scratch/extending-cookbook/httpd/resources
  * template[/Users/franklinwebber/scratch/extending-cookbook/httpd/resources/vhost.rb]
action create
    - create new file /Users/franklinwebber/scratch/extending-
cookbook/httpd/resources/vhost.rb
    - update content in file /Users/franklinwebber/scratch/extending-
cookbook/httpd/resources/vhost.rb from none to e3b0c4
    (diff output suppressed by config)
  * directory[/Users/franklinwebber/scratch/extending-cookbook/httpd/providers] action create
```


The chef command-line tool allows you to generate some initial directories and resource file. While we are developing a Custom Resource the former name for them was called Light Weight Resource Provider or LWRP. The chef command still uses the acronym lwrp as the generate sub-command.

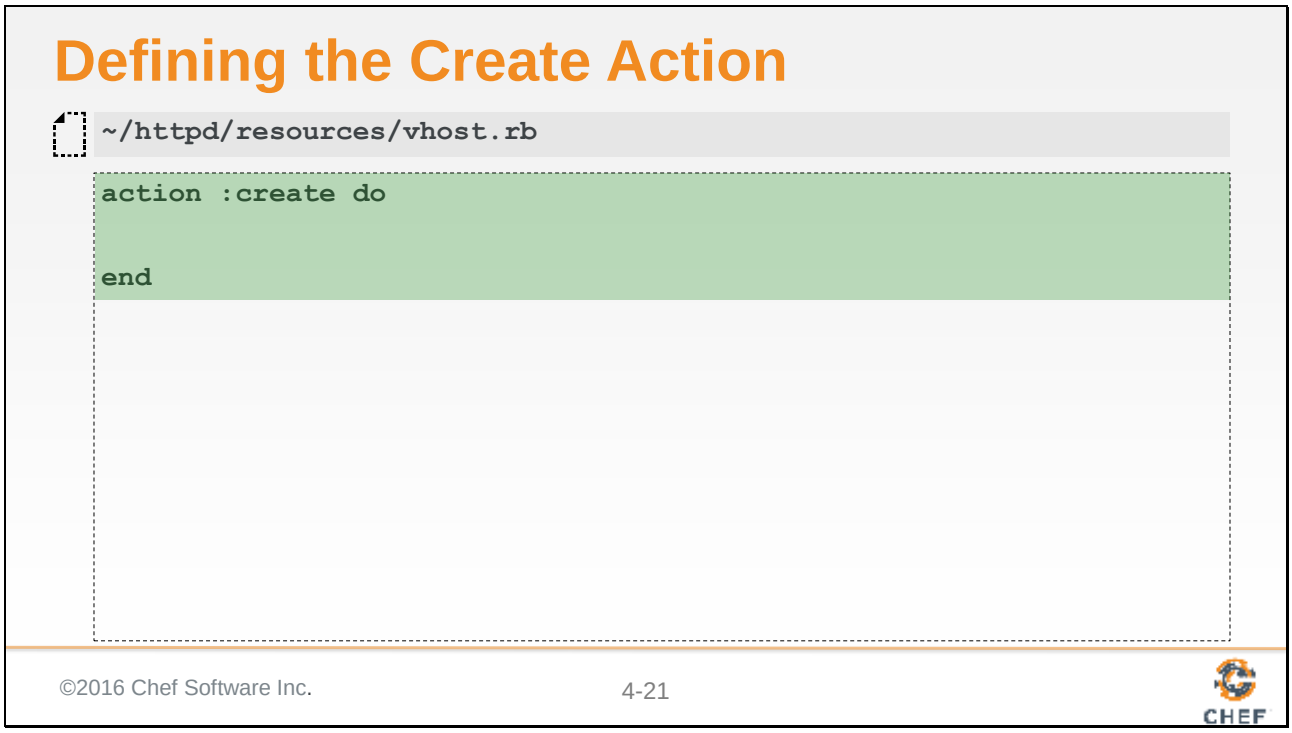
We call these multiple different sites, available on different ports, a virtual host. This is often abbreviated as 'vhost'. Create a custom resource with the name 'vhost'.

Slide 20

A LWRP (light-weight resource provider) requires two directories. A resources directory and a provider directory. The custom resource implementation requires only the resources directory.

The providers directory is not needed so it should be removed.

Slide 21



Within the resources directory a file named 'vhost' should exist. Within it we are simply going to define an action with the name :create. This create action is where we will define the resources necessary to create a new vhost.

Slide 22

Implementing the Create Action

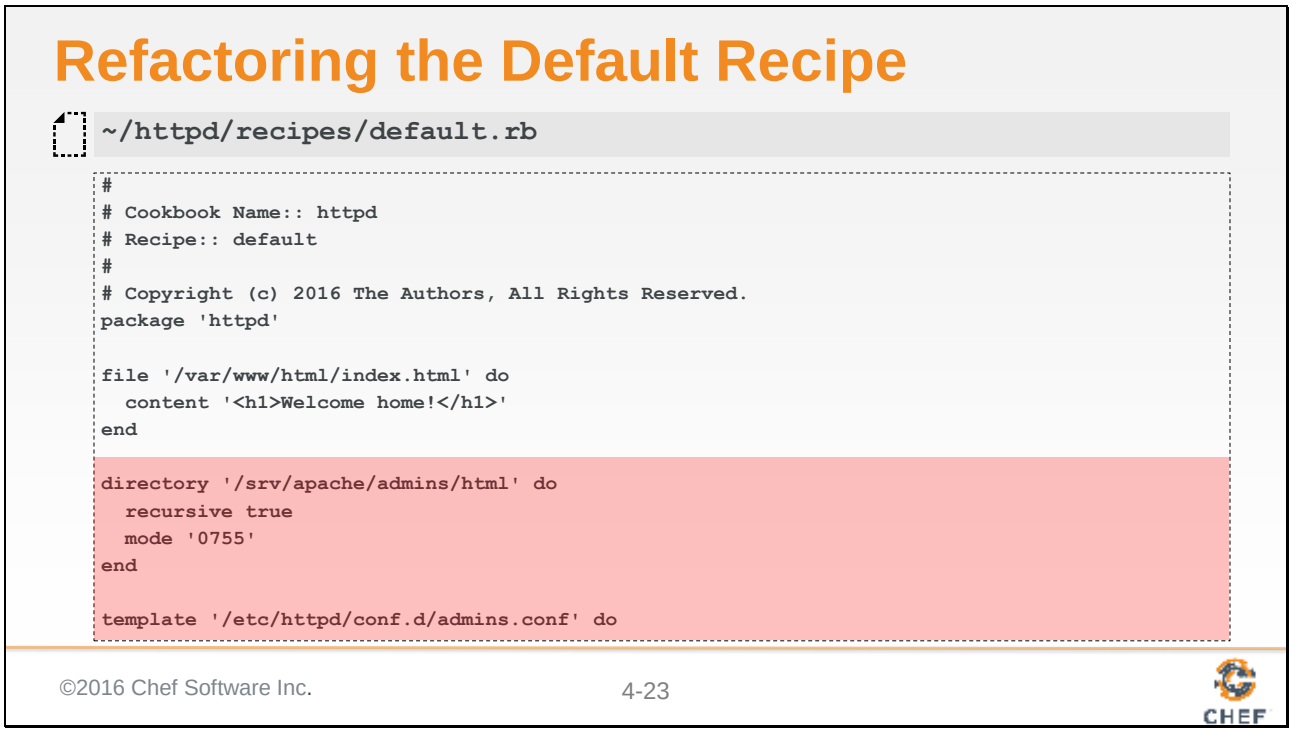
~/httpd/resources/vhost.rb

```
action :create do
  directory '/srv/apache/admins/html' do
    recursive true
    mode '0755'
  end

  template '/etc/httpd/conf.d/admins.conf' do
    source 'conf.erb'
    mode '0644'
    variables(document_root: '/srv/apache/admins/html',port: 8080)
    notifies :restart, 'service[httpd]'
  end

  file '/srv/apache/admins/html/index.html' do
    content '<h1>Welcome admins!</h1>'
  end
end
```


To create a new virtual host we need to generate a directory, add a configuration file, and define an html file. This is similar to the exact same resources that we defined for the admin site in the default recipe.

Our first implementation for our custom resource will create the exact same admin site exactly as it is done in the default recipe. These values are hard-coded to the admin site which we will address after getting our implementation working.

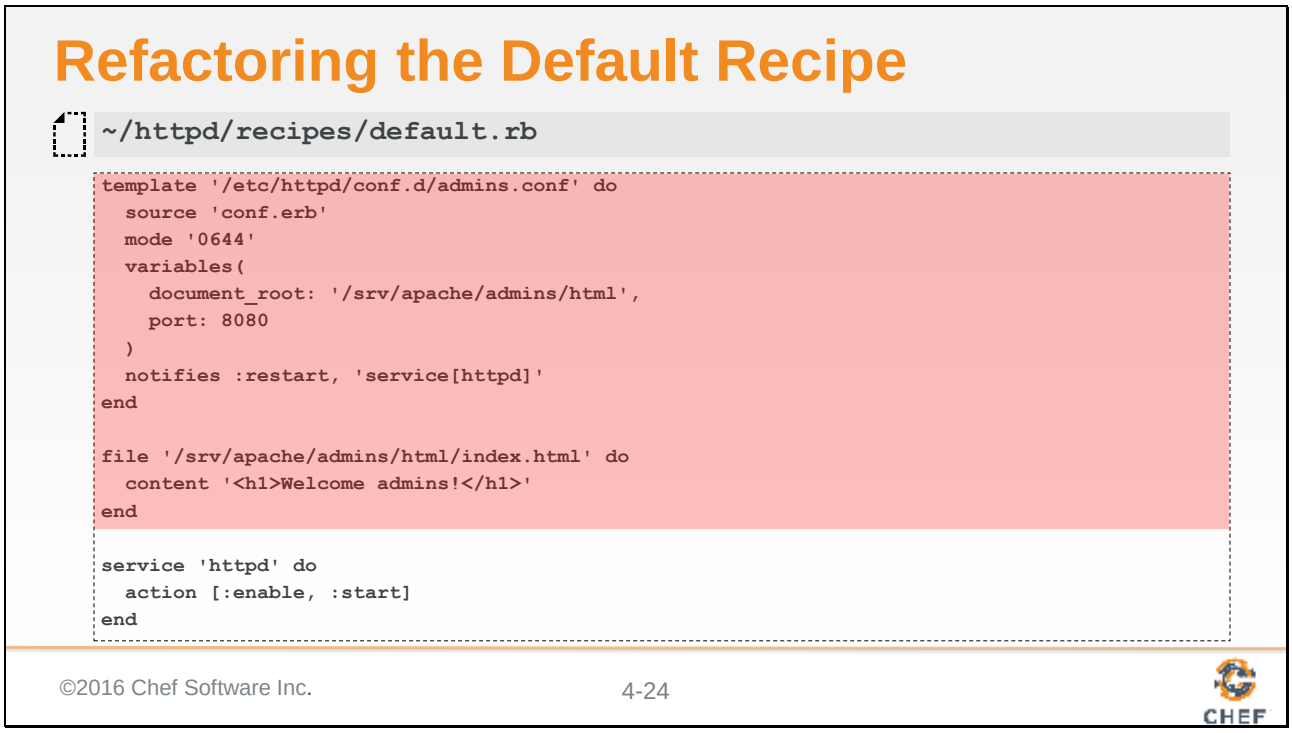
Slide 23



Now that those three resources are defined within the custom resource we want to use it within our recipe. We can now remove the use of these three resources within the default recipe.

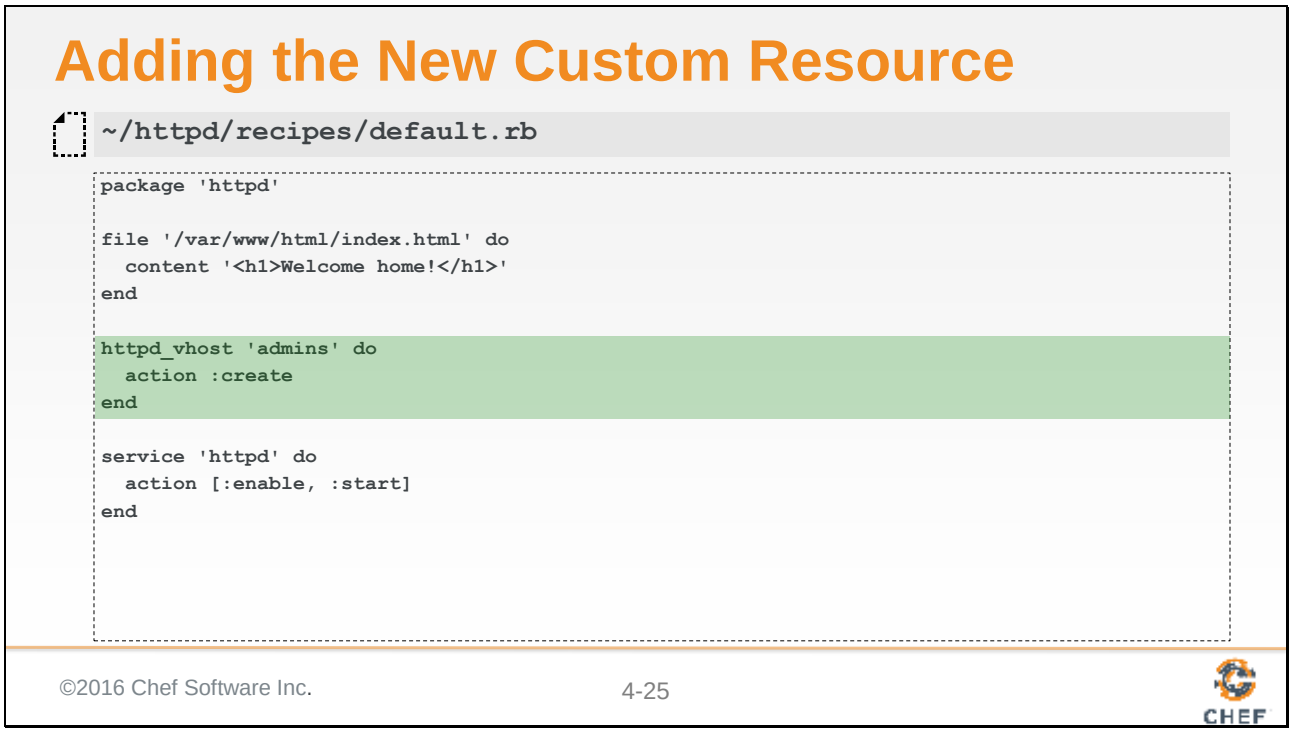
Remove the directory resource, the template resource, and the file resource that generate the admin site.

Slide 24



Remove the directory resource, the template resource, and the file resource that generate the admin site.

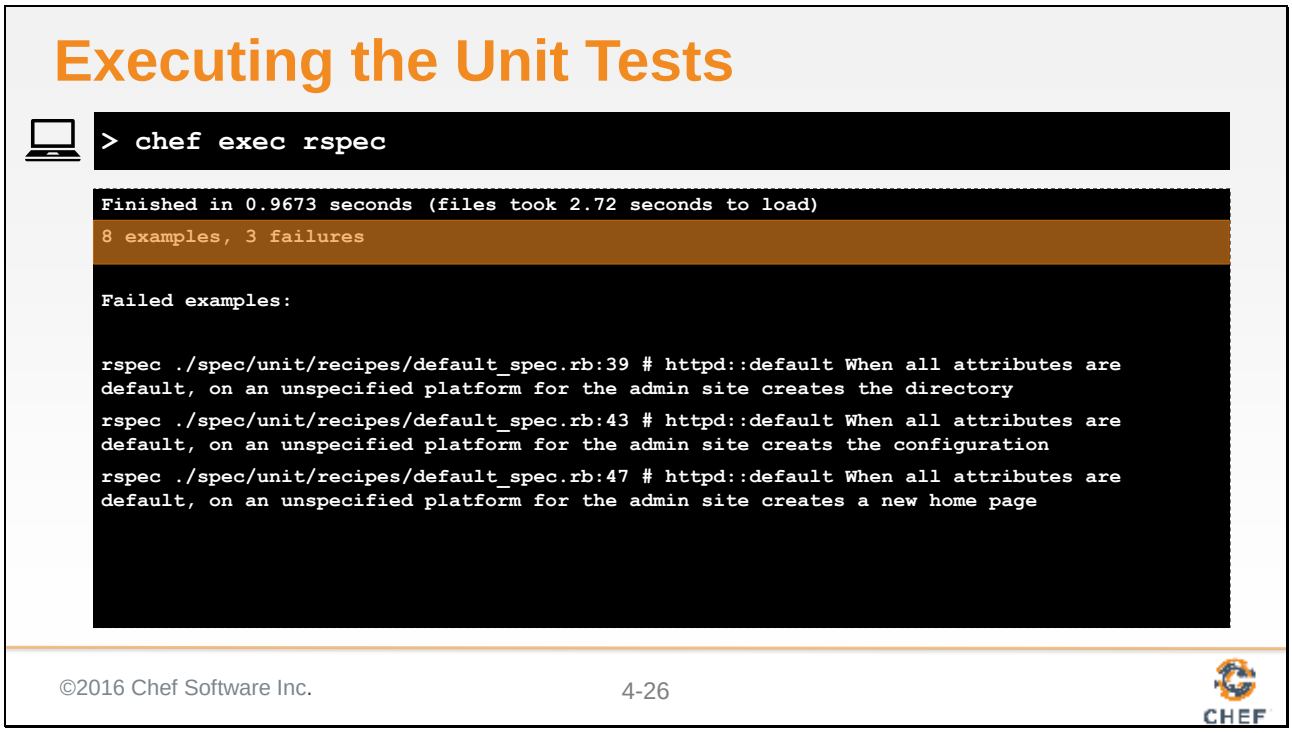
Slide 25



Now we can insert the custom resource that is create for us. The full name of the custom resource comes from the name of the cookbook joined with an underscore to the name of the ruby file defined within the resources directory.

In this instance the cookbook's name is 'httpd' and the ruby file is named 'vhost' so the default name for the resource is 'httpd_vhost'. We inform the resource that we want to generate the site for admins, though the name of the resource is not used in any way in our definition. We explicitly state that the resource will use the create action.

Slide 26

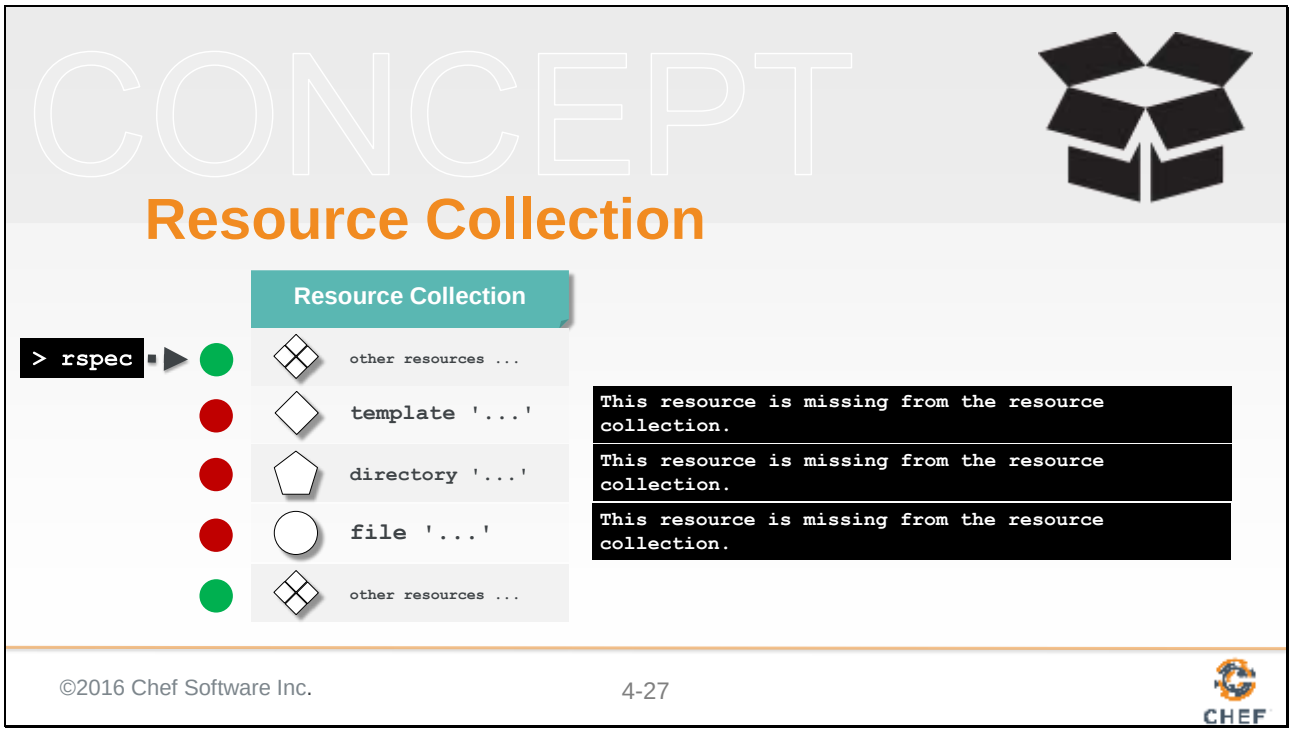


With the custom resource defined now within the default recipe it is time to run our unit tests to ensure that we have not broken our implementation.

When executing the tests you will see three failures. These three failures will instruct you that it does not see the following resources created: the directory for the admin site; the configuration file built from the template; and the html file.

This does not seem right. The resources defined within the custom resource do just that.

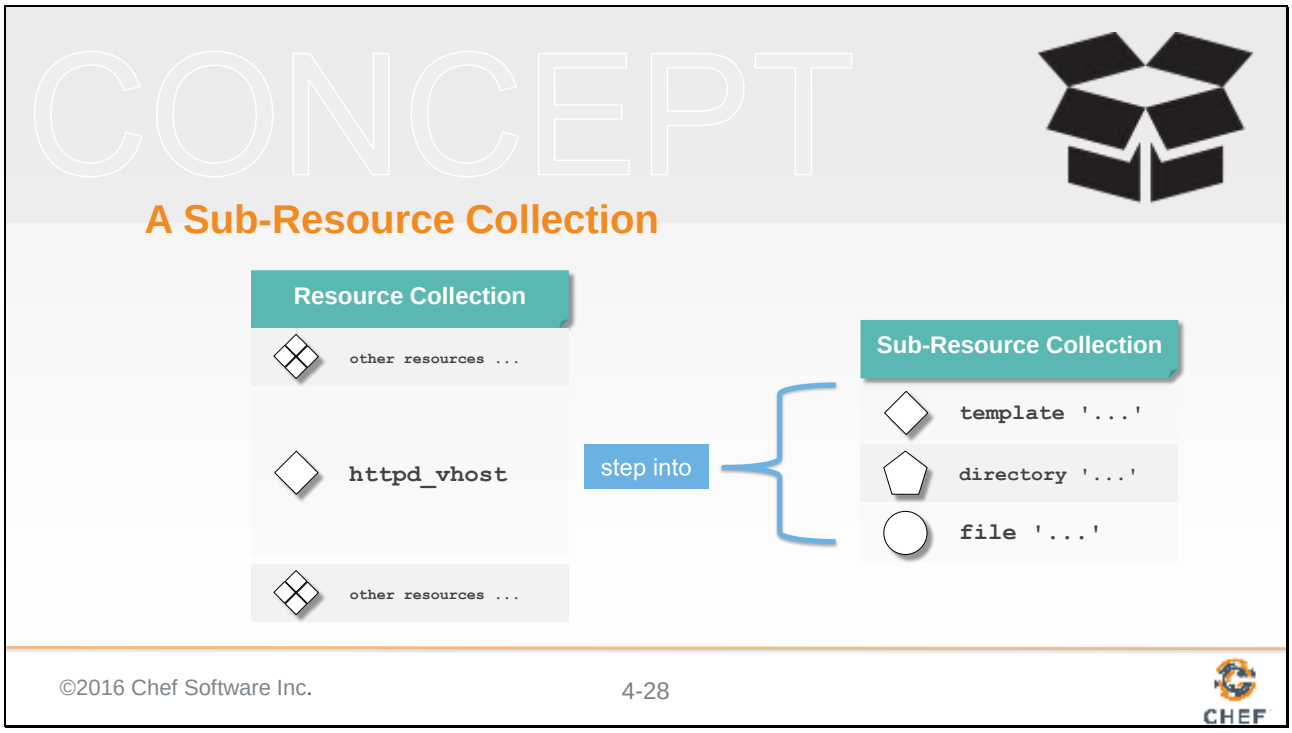
Slide 27



Remember the ChefSpec expectations are validating the contents of the state of the resource collection.

When we created this custom resource we moved the three resources within the recipe into the action we defined. This changed the state of the resource collection and caused the failures we see when we execute the test suite.

Slide 28



This custom resource created a resource collection within our resource collection; a sub-resource collection. ChefSpec by default does not step into this sub-resource collection. We can however enable that behavior if we modify our test setup to explicitly state we are interested in evaluating the contents of this sub-resource collection.

We will discuss more about the implications of having a sub-resource collection in the follow-up module.

Slide 29

Updating the Unit Tests to Verify Custom Resource

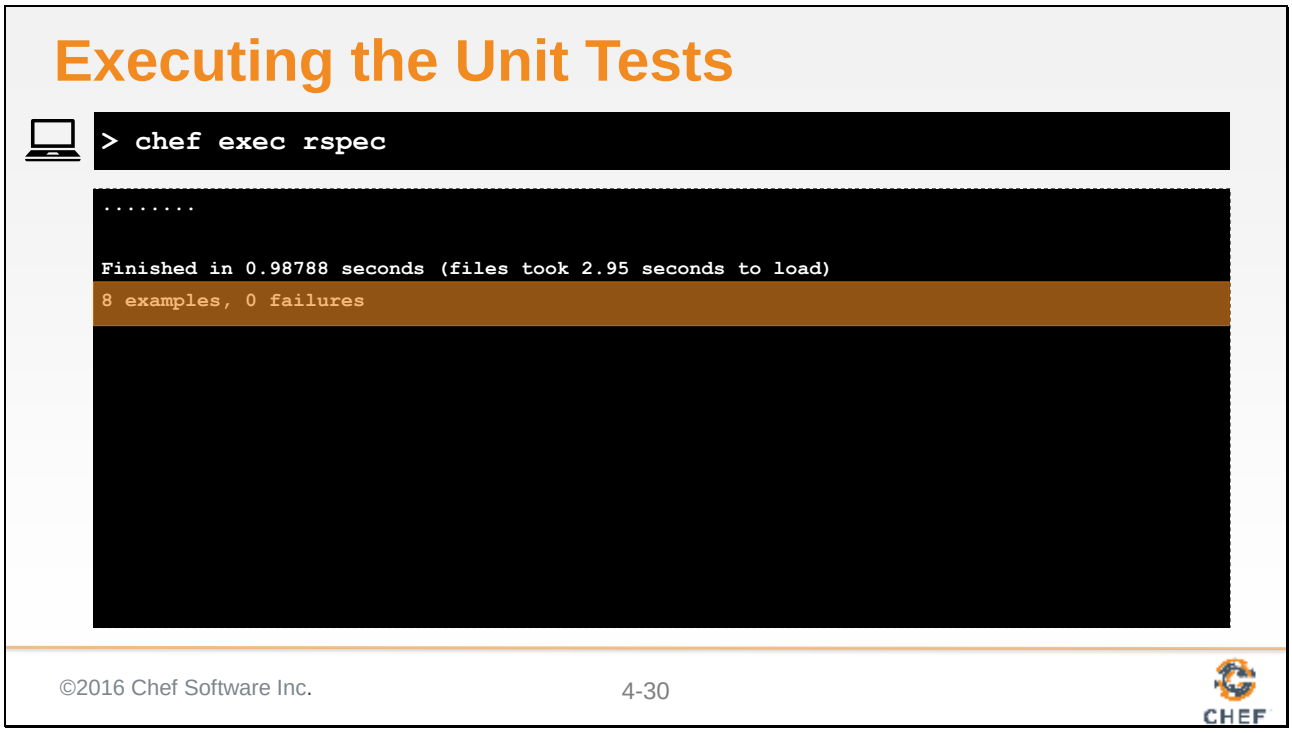
~/httpd/spec/unit/recipes/default_spec.rb

```
require 'spec_helper'

describe 'httpd::default' do
  context 'When all attributes are default, on an unspecified platform' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(step_into: ['httpd_vhost'])
      runner.converge(described_recipe)
    end

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

# ... CONTINUES ON THE NEXT SLIDE ...
```

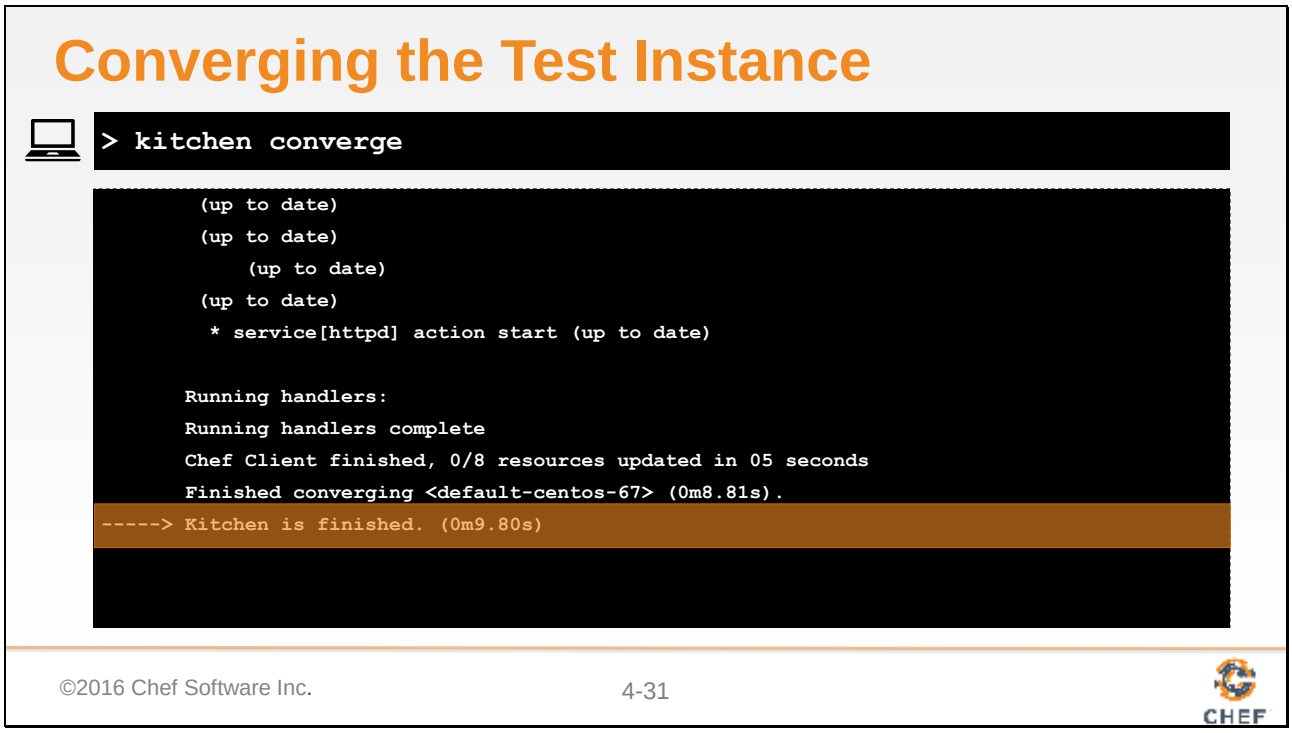

The unit tests fail because the resources defined within the custom resource are now no longer placed onto the resource collection. This is because the custom resource is placed on the resource collection and the resources internally within it are placed on a secondary resource collection that the custom resource owns.

To ask our unit tests to verify the resources defined within our custom resource we need to explicitly ask the ChefSpec runner to step into the resource and examine the resources it uses to accomplish it's work.

Slide 30

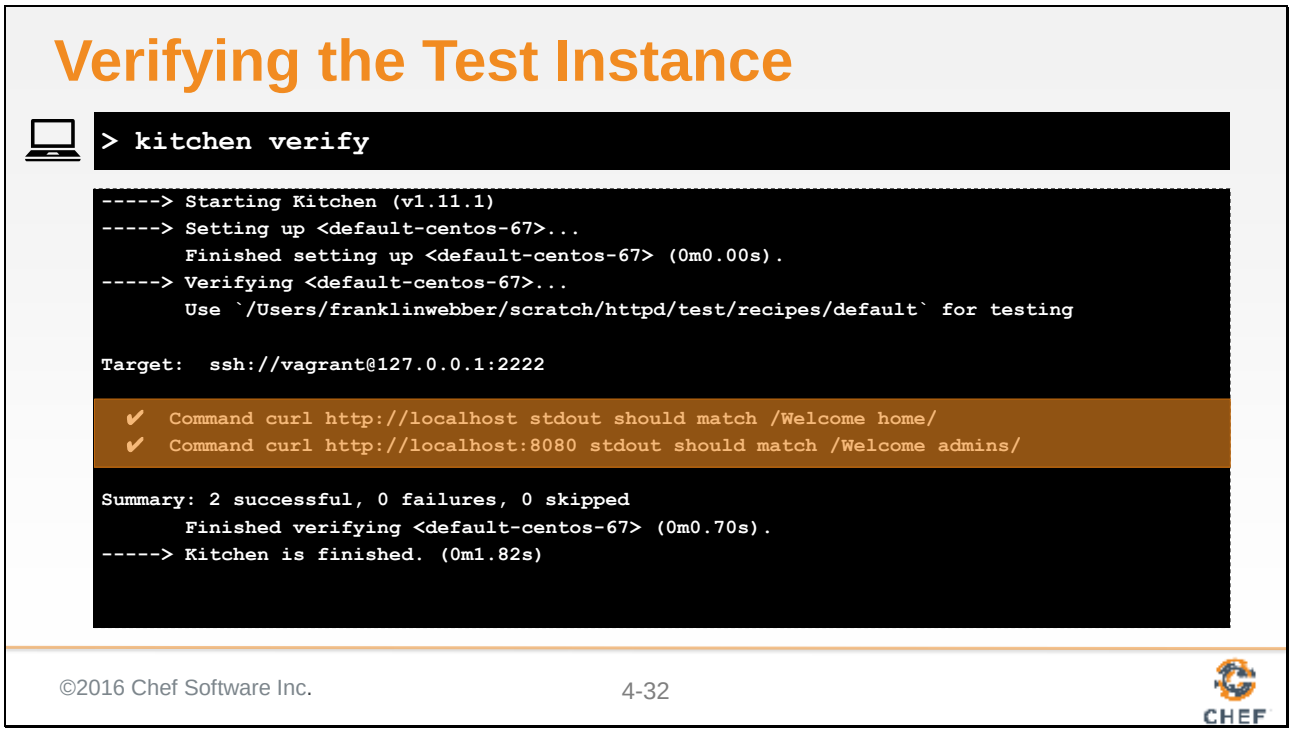


Running the unit tests again should show all the expectations have been met.



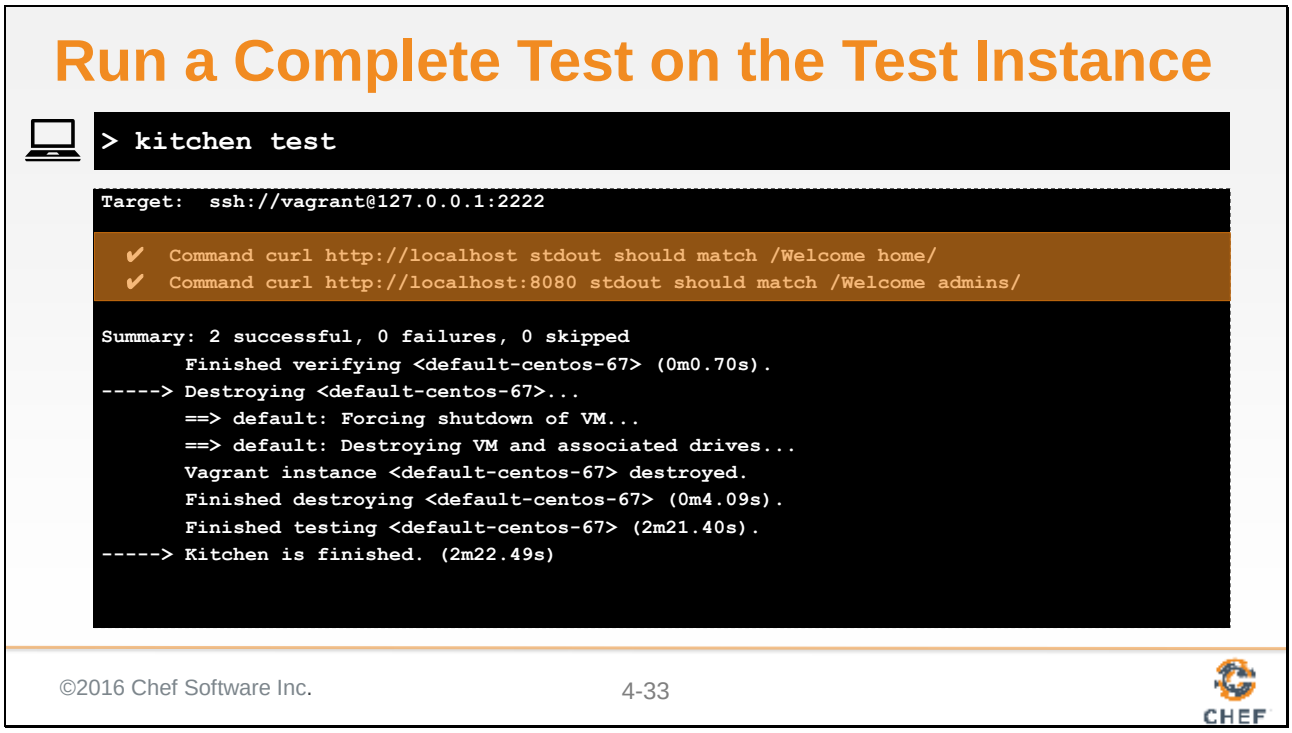
It is also important to execute the integration tests defined. First converging the test instance to ensure the recipe is defined correctly and converges successfully.

Slide 32



And finally we verify that the state of the system is still hosting our two sites.

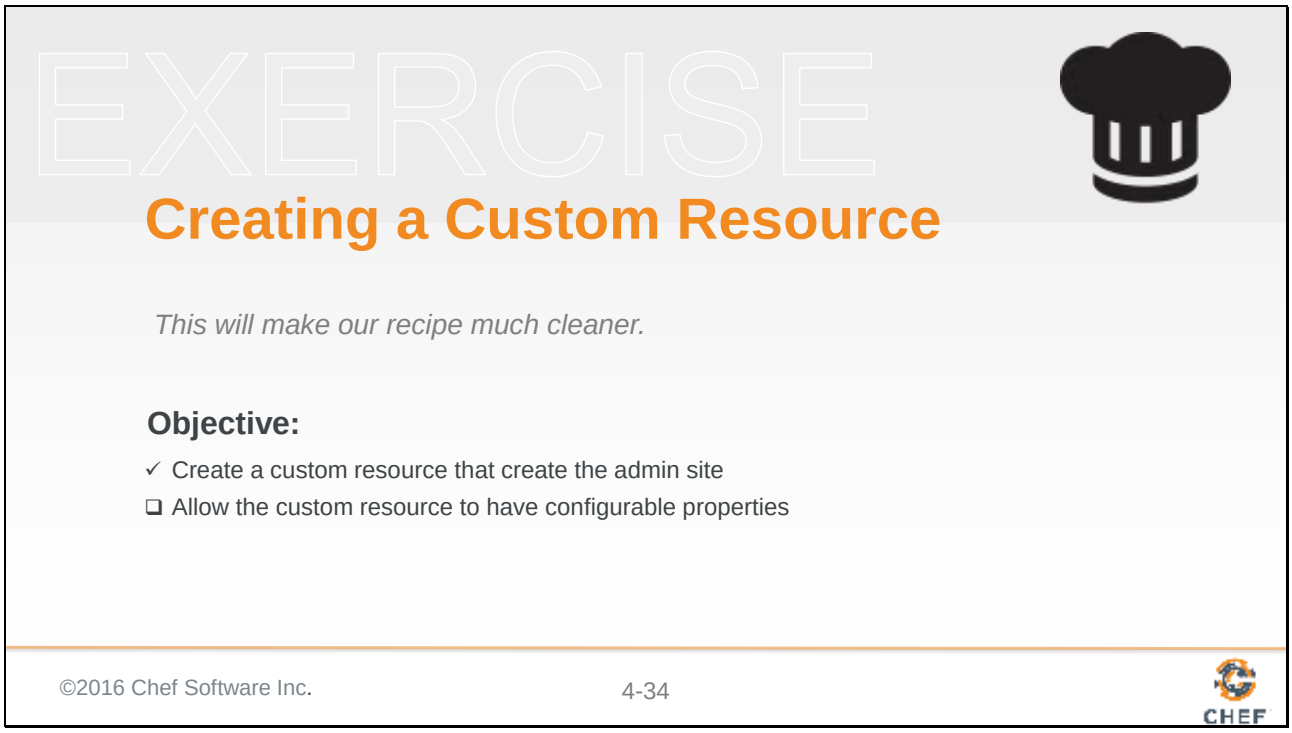
Slide 33



We made changes to the recipe and then used kitchen to converge this recipe against the test instance. This ensured that our recipe will successfully converge against a system that has already been configured and not raise any errors.

However, we still need to ensure that the recipe will converge successfully on a brand new instance so it is important to ask Test Kitchen to destroy the instance, converge a new instance, and verify the results. This can be done with the 'kitchen test' command.

Slide 34

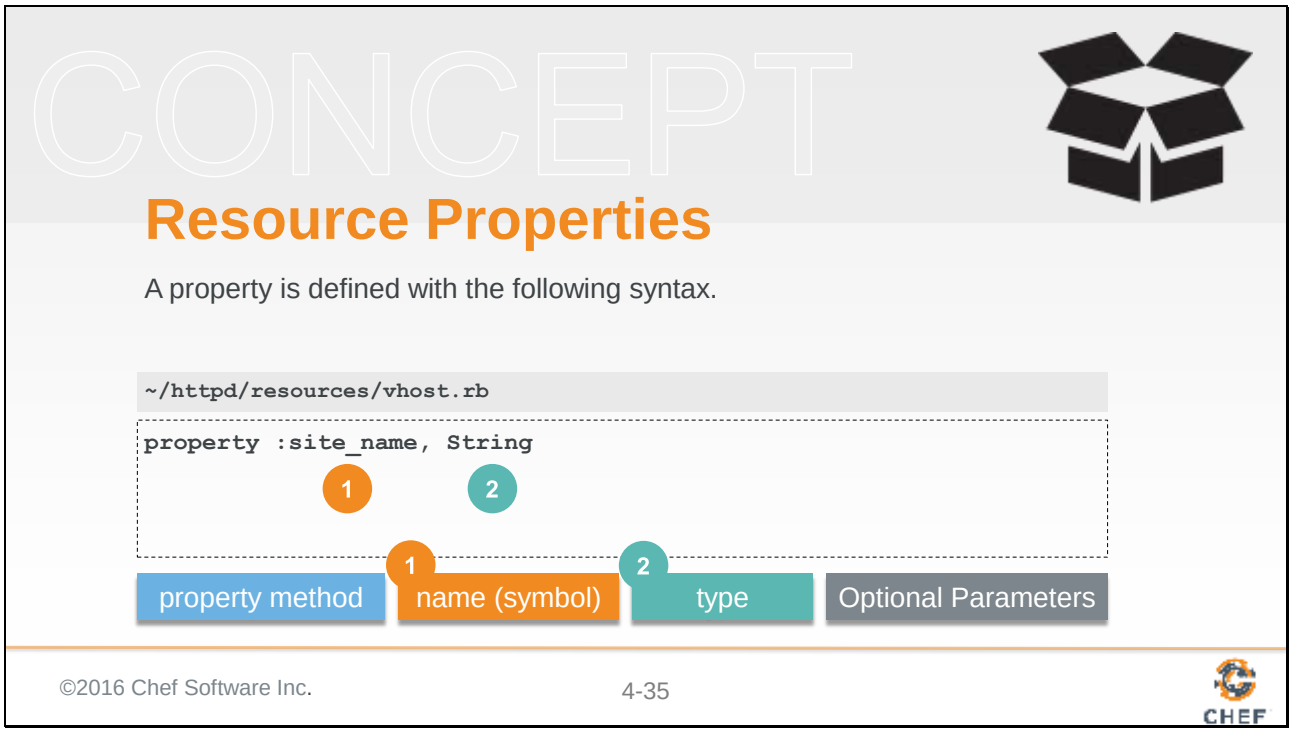


The initial implementation of the custom resource has been created and we have verified that it works by running our two test suites.

Now it is time to address the problem with the implementation having hard-coded values specific to the admin site. We want to make it more generic so that it can deploy a different, custom site for us if needed.

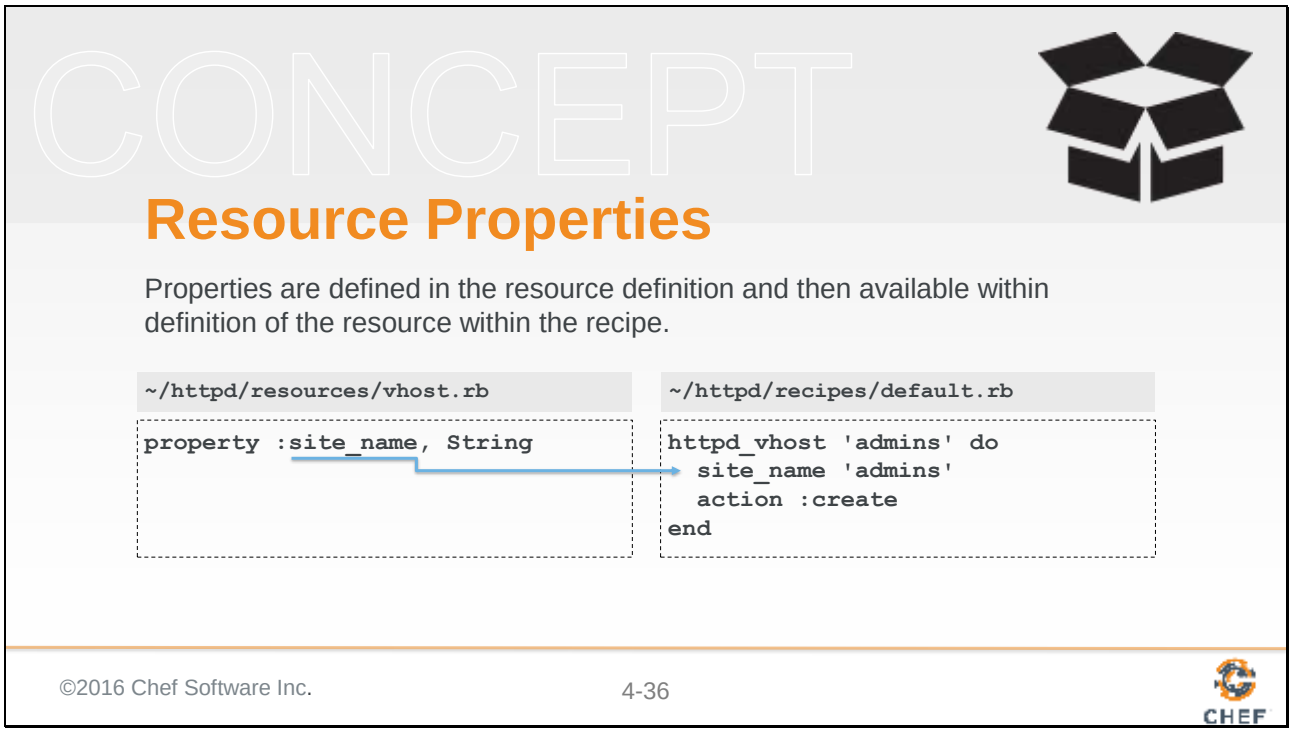
This can be done through properties that you defined on the custom resource.

Slide 35



Properties are defined in the same file as you define the resource actions. Generally these are defined at the top of the file to make them immediately visible. A property is defined by specifying a method named property with two required parameters and a third set of optional parameters. The name of the property is defined as a Ruby Symbol. The type is a Ruby class name. This type enforces what kind of values are supported by this property; typically it is a String for text and a Fixnum for numbers. The optional parameters are defined as a Hash. We will explore defining a property with these parameters in the next module.

Slide 36



The property that you define within the custom resource definition becomes part of how you can describe the resource within the recipe.

Defining a Property to Manage the site_name

~/httpd/resources/vhost.rb

```
property :site_name, String

action :create do

  directory "/srv/apache/#{site_name}/html" do
    recursive true
    mode '0755'
  end

  template "/etc/httpd/conf.d/#{site_name}.conf" do
    source 'conf.erb'
    mode '0644'
    variables(document_root: "/srv/apache/#{site_name}/html", port: 8080)
    notifies :restart, 'service[httpd]'
  end

  # ... CONTINUES ON THE NEXT SLIDE ...
```


Let's start by defining a property named 'site_name' that will contain the name of the site we want to create. The name of the site will be used to create the directory for our index page, the configuration file details, and the message we send out to the visitor.

The 'site_name' is going to be a text so we specify the type as String.

Updating the Action to use the Property

~/httpd/resource/vhost.rb

```
action :create do
  directory "/srv/apache/#{site_name}/html" do
    recursive true
    mode '0755'
  end

  template "/etc/httpd/conf.d/#{site_name}.conf" do
    source 'conf.erb'
    mode '0644'
    variables(document_root: "/srv/apache/#{site_name}/html", port: 8080)
    notifies :restart, "service[httpd]"
  end

  file "/srv/apache/#{site_name}/html/index.html" do
    content "<h1>Welcome #{site_name}!</h1>"
  end
end
```


Within the action implementation we want to remove the mention of 'admin' and replace it with the value found within the 'site_name' custom property. A resource property creates a method with the same name as the property.

Now we need to replace the 'admin' text with the result of the property. This requires us to update a number of our resources to use String interpolation to express the directory created, the configuration path, and then default html page.

Slide 39

Updating the Resource to use the Property

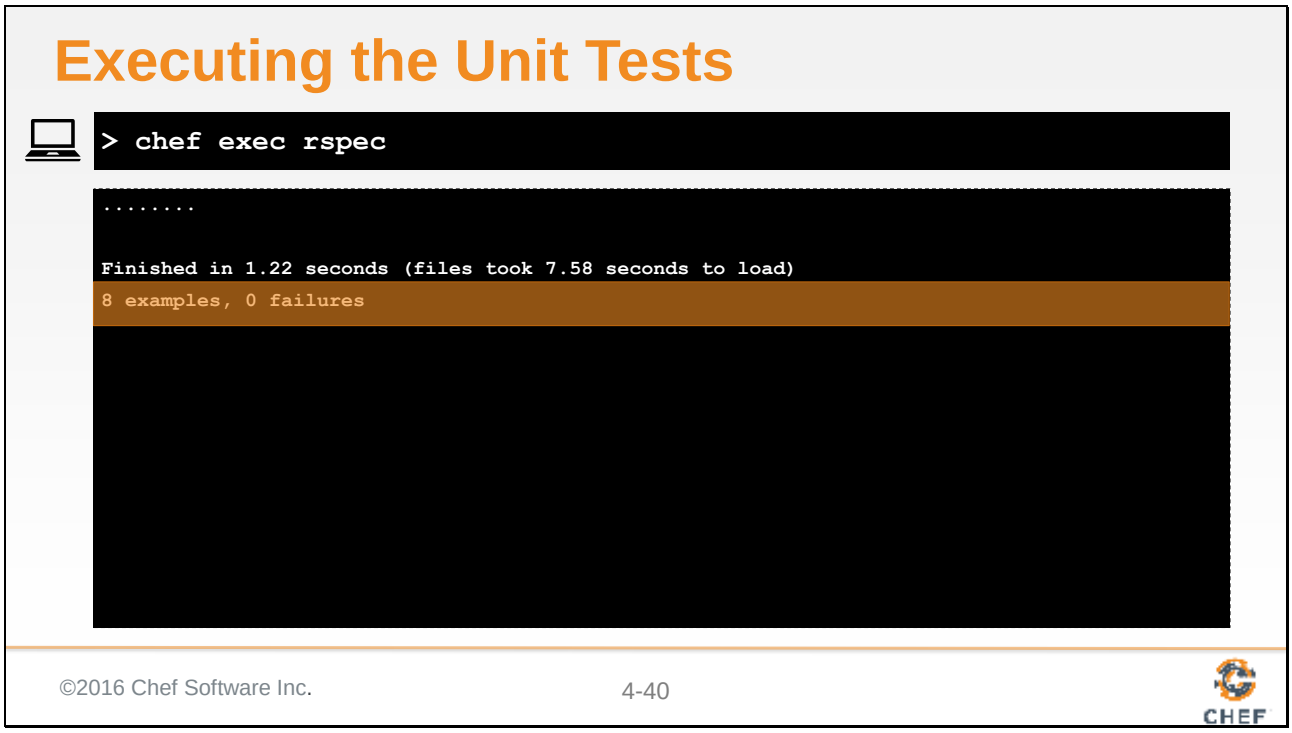
~/httpd/recipes/default.rb

```
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome home!</h1>'
end

httpd_vhost 'admins' do
  site_name 'admins'
  action :create
end

service 'httpd' do
  action [:enable, :start]
end
```


This property is not automatically defined and does not contain a default value so we must add this propery to the custom resource implementation with the default recipe. In this case we are adding 'site_name' and specifying the value is 'admins'.

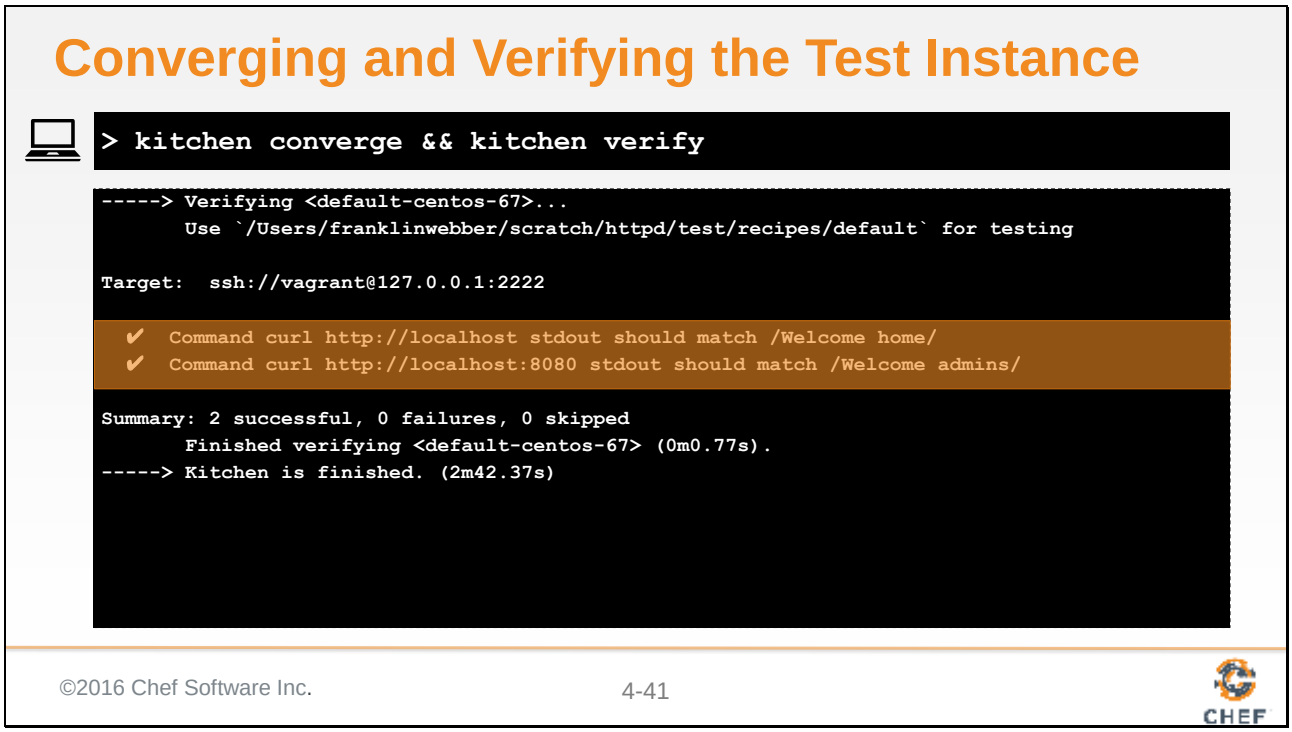
This means our implementation should be exactly the same as before but the details are now configurable through this property instead of being hard-coded to 'admin'.

Slide 40



The unit tests should pass when executed.

Slide 41



The integration tests should pass when executed.

Slide 42

EXERCISE

Creating a Custom Resource

That's much better.

Objective:

- ✓ Create a custom resource that create the admin site
- ✓ Allow the custom resource to have configurable properties

CHEF

We now have a custom resource implementation that has helped express our intentions more clearly in the default recipe.

Slide 43

LAB

httpd_vhost - site_port Property

- Create a custom resource property named **site_port** that is a *Fixnum*
- Within the httpd_vhost's create action replace the hard-coded value 8080 with the **site_port** property
- Within the default recipe set the httpd_vhost resource named 'admins' to have a site_port 8080

CHEF

The custom resource still needs a little more work to make it configurable. The create action is still hard-coded to specify the port 8080 for all sites that are created.

During this exercise you will define a new property within the custom resource that allows a port to be specified for the site. Replace any hard-coded port values within the resource action implementation and then add the new property to the implementation of the custom resource in the default recipe.

Defining a Property to Manage the site_port

~/httpd/resource/vhost.rb

```
property :site_name, String
property :site_port, Fixnum

action :create do
  directory "/srv/apache/#{site_name}/html" do
    recursive true
    mode '0755'
  end

  template "/etc/httpd/conf.d/#{site_name}.conf" do
    source 'conf.erb'
    mode '0644'
    variables(document_root: "/srv/apache/#{site_name}/html", port: site_port)
    notifies :restart, 'service[httpd]'
  end
# ... REMAINDER OF CUSTOM RESOURCE ...
```


The property is defined near the top of the resource file. A port is generally a whole number so we want that reflected in the type.

A Fixnum can contain negative integers and floating point numbers so this type does not perfectly represent the domain of acceptable values. Later we may explore ways to ensure better restrictions on the values provided to properties.

Within the action implementation the 8080 value should be replaced with the value found in 'site_port'.

Slide 45

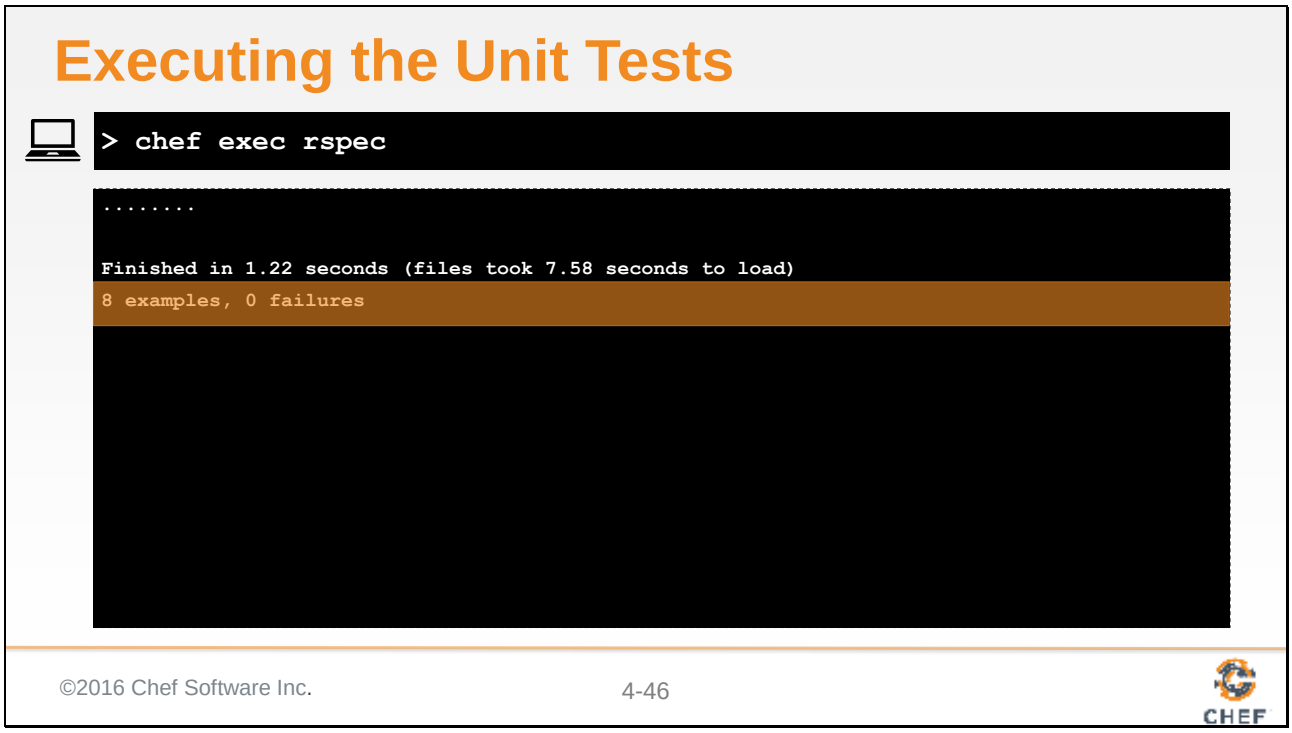
Updating the Resource to use the Property

~/httpd/recipes/default.rb

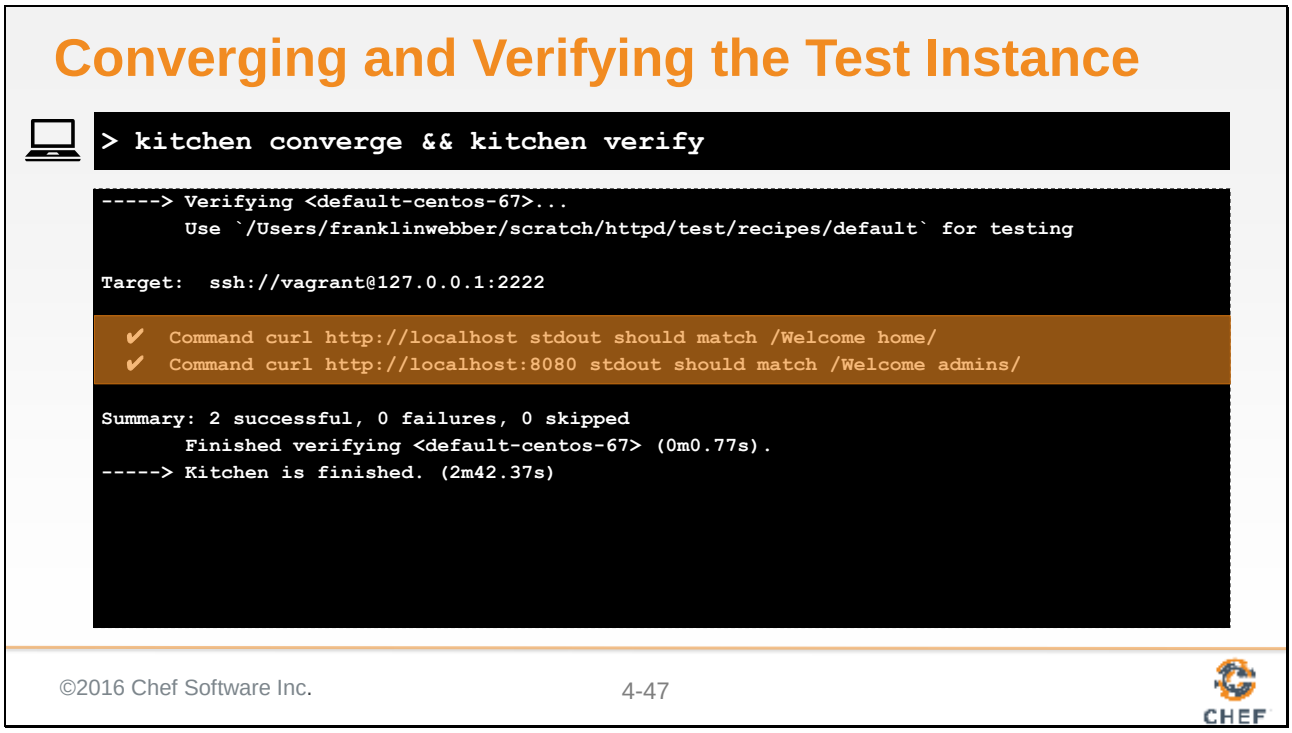
```
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome home!</h1>'
end

httpd_vhost 'admins' do
  site_name 'admins'
  site_port 8080
  action :create
end

service 'httpd' do
  action [:enable, :start]
end
```


In the default recipe, within the 'httpd_vhost' resource, we must define a value for this site_port. Similar to before we simply define the value that was previously hard-coded here as a property.

Slide 46



The unit tests should pass when executed.



The integration tests should pass when executed.

Slide 48

LAB

httpd_vhost - site_port Property

- ✓ Create a custom resource property named **site_port** that is a *Fixnum*
- ✓ Within the httpd_vhost's create action replace the hard-coded value 8080 with the **site_port** property
- ✓ Within the default recipe set the httpd_vhost resource named 'admins' to have a site_port 8080

CHEF

With the site_port property developed the 'httpd_vhost' custom resource is now capable of being used to create more sites if needed for different roles on different ports for our web server.

Slide 49

By default Apache creates a welcome configuration file within the same directory we are creating our new virtual hosts. We want to delete this configuration file but we want to create a resource that will also cleanup any html files that our resource might create as well. This will allow us to create and remove sites as we want.

Slide 50

LAB

httpd_vhost Remove Action

- Define the 'remove' action for httpd_vhost that defines the policy:
  - **A directory named "/srv/apache/#{site_name}/html" is deleted**
  - **A file named "/etc/httpd/conf.d/#{site_name}.conf" is deleted**
- Update the default recipe's policy to include a new resource:
  - **An httpd_vhost resource named 'welcome' is removed**

CHEF

This next lab exercise challenges you to create the remove action for the custom resource, use that remove action to remove the default site that ships with the webserver, and deploy a new site instead which welcomes users.

Slide 51

Defining the Resource's Remove Action

~/httpd/resources/vhost.rb

```
# ... CREATE ACTION ...

action :remove do
  directory "/srv/apache/#{site_name}/html" do
    action :delete
  end

  file "/etc/httpd/conf.d/#{site_name}.conf" do
    action :delete
  end
end
```


The remove action asks that you remove the directory that may or may not exist at the location dependent on the site_name provided as a property. We also want it to remove the configuration file from the webserver's default configuration directory.

Slide 52

Adding the Resource with Remove Action to the Recipe

~/httpd/recipes/default.rb

```
package 'httpd'

httpd_vhost 'welcome' do
  site_name 'welcome'
  action :remove
end

httpd_vhost 'admins' do
  site_port 8080
  site_name 'admins'
  action :create
end

service 'httpd' do
  action [:enable, :start]
end
```


When httpd initial sets itself up it deploys the first, default site, with a welcome configuration file that we want to remove. While the 'welcome' directory does not exist the configuration file does and so we want that removed from the system.

This will ensure the default site that is deployed on port 80 is no longer deployed.

Slide 53

LAB

httpd_vhost Remove Action

- ✓ Define the 'remove' action for httpd_vhost that defines the policy:
  - **A directory named "/srv/apache/#{site_name}/html" is deleted**
  - **A file named "/etc/httpd/conf.d/#{site_name}.conf" is deleted**
- ✓ Update the default recipe's policy to include a new resource:
  - **An httpd_vhost resource named 'welcome' is removed**

CHEF

The resource now has two actions and we have removed the initial welcome site.

Slide 54

LAB

httpd_vhost Remove Action

- Update the default recipe's policy:
  - Remove the file resource named '/var/www/html/index.html'
  - Add an httpd_vhost resource named 'users' is created with the site_port 80
- Update the ChefSpec tests to stop expecting the file resource and start expecting the new resources found within the httpd_vhost resource named 'users'
- Update the InSpec tests to expect the default site to "Welcome users!"

CHEF

This next lab exercise challenges you to create the remove action for the custom resource, use that remove action to remove the default site that ships with the webserver, and deploy a new site instead which welcomes users.

Slide 55

Removing the Resource from the Recipe

~/httpd/recipes/default.rb

```
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome home!</h1>'
end

httpd_vhost 'admins' do
  site_port 8080
  site_name 'admins'
  action :create
end

service 'httpd' do
  action [:enable, :start]
end
```


The file resource within the default recipe modifies a generic files that httpd deploys. Manipulating this resource is no longer important so we want to remove this resource.

Slide 56

Adding the Resource to create the users site

~/httpd/recipes/default.rb

```
httpd_vhost 'welcome' do
  site_name 'welcome'
  action :remove
end

httpd_vhost 'users' do
  site_port 80
  site_name 'users'
  action :create
end

httpd_vhost 'admins' do
  notifies :restart, 'service[httpd]'
end
```


Now we want to add in our users site with our custom resource. We define a site_port and and site_name to ensure we receive the correct message on the correct port.

Slide 57

Removing the Un-needed Unit Test Expectation

~/httpd/spec/unit/recipes/default_spec.rb

```
# ... EXAMPLES DEFINED ABOVE ...

describe 'for the default site' do
  it 'writes out a new home page' do
    expect(chef_run).not_to
render_file('/var/www/html/index.html').with_content('<h1>Welcome home!</h1>')
  end
end

# ... EXAMPLES DEFINED BELOW ...
```


CHEF

This changes our default expectations that the site will say welcome home so we want to remove that content from our unit test.

Slide 58

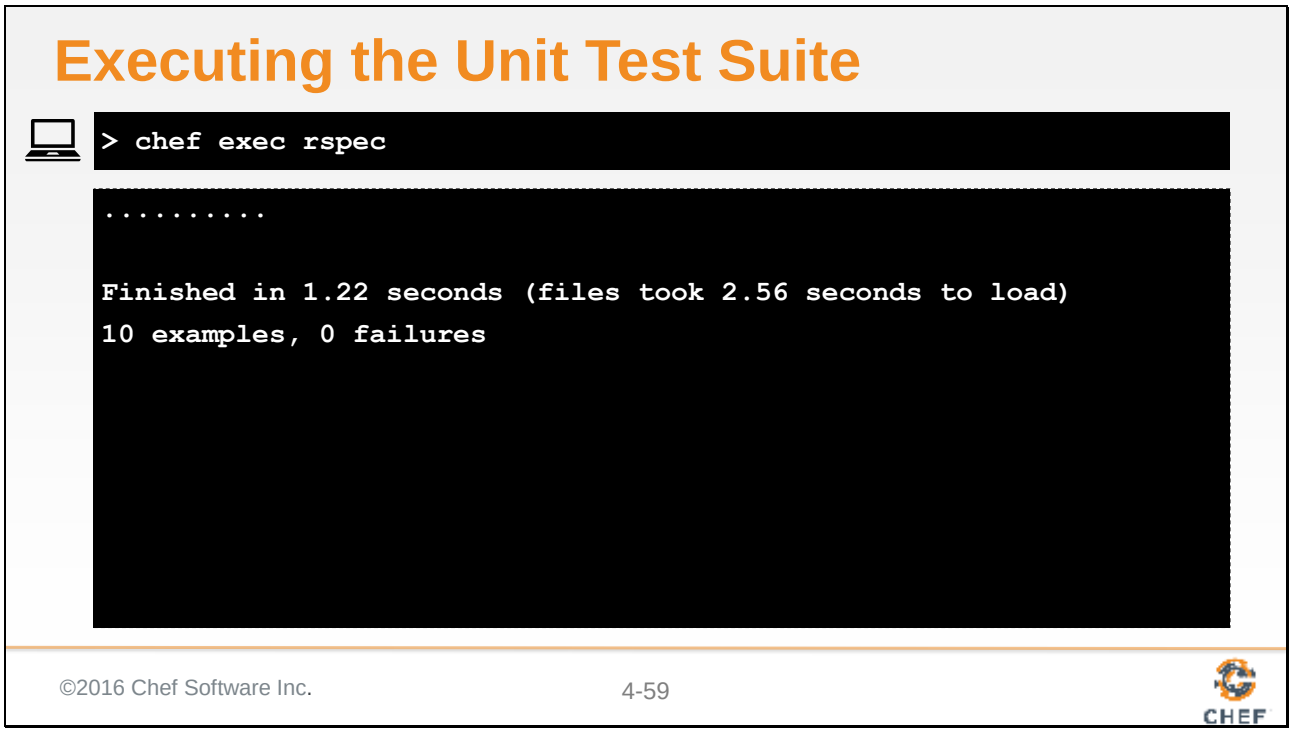
Adding Expectations for the users Site

~/httpd/spec/unit/recipes/default_spec.rb

```
 # ... EXAMPLES DEFINED ABOVE ...
  describe 'for the users site' do
    it 'creates the directory' do
      expect(chef_run).to create_directory('/srv/apache/users/html')
    end

    it 'creates the configuration' do
      expect(chef_run).to render_file('/etc/httpd/conf.d/users.conf')
    end

    it 'creates a new home page' do
      expect(chef_run).to
render_file('/srv/apache/users/html/index.html').with_content('<h1>Welcome users!</h1>')
    end
  # ... EXAMPLES DEFINED ABOVE ...
```

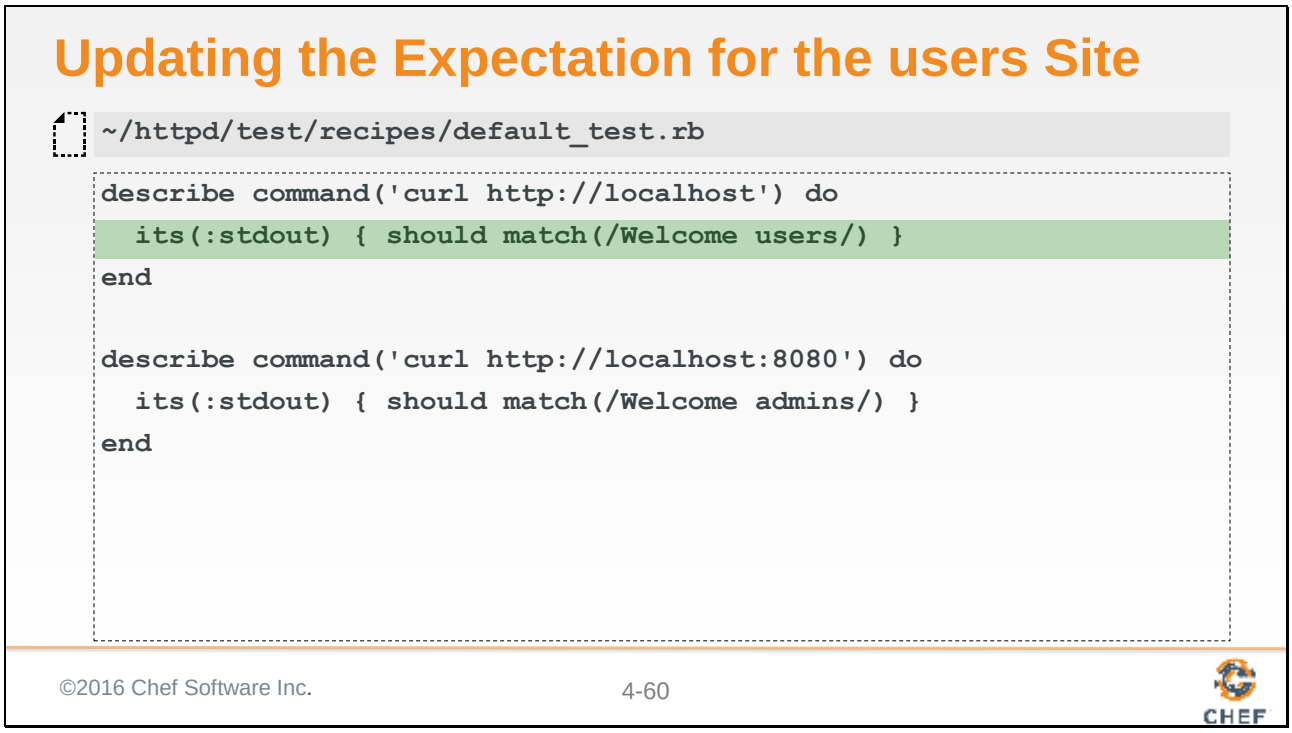

And add a new series of expectations that are very similar to the admins site. We want to ensure that the following are created: a directory to store the html; a configuration file for users; and a new html file that contains a message for the users.

Slide 59



Executing the unit tests we should see all these brand new expectations passing.

Slide 60



Finally we want to change the integration test to verify the message on port 80 to welcome users and not to welcome visitors home.

Slide 61

Converging and Verifying the Test Instance

```
> kitchen converge && kitchen verify
```

```
-----> Verifying <default-centos-67>...
       Use `/Users/franklinwebber/scratch/httpd/test/recipes/default` for testing

Target:  ssh://vagrant@127.0.0.1:2222

  ✔  Command curl http://localhost stdout should match /Welcome home/
  ✔  Command curl http://localhost:8080 stdout should match /Welcome admins/

Summary: 2 successful, 0 failures, 0 skipped
       Finished verifying <default-centos-67> (0m0.77s).
-----> Kitchen is finished. (2m42.37s)
```


Executing the integration tests should result in all examples passing successfully.

Slide 62

LAB

httpd_vhost Remove Action

- ✓ Update the default recipe's policy:
  - Remove the file resource named '/var/www/html/index.html'
  - Add an httpd_vhost resource named 'users' is created with the site_port 80
- ✓ Update the ChefSpec tests to stop expecting the file resource and start expecting the new resources found within the httpd_vhost resource named 'users'
- ✓ Update the InSpec tests to expect the default site to "Welcome users!"

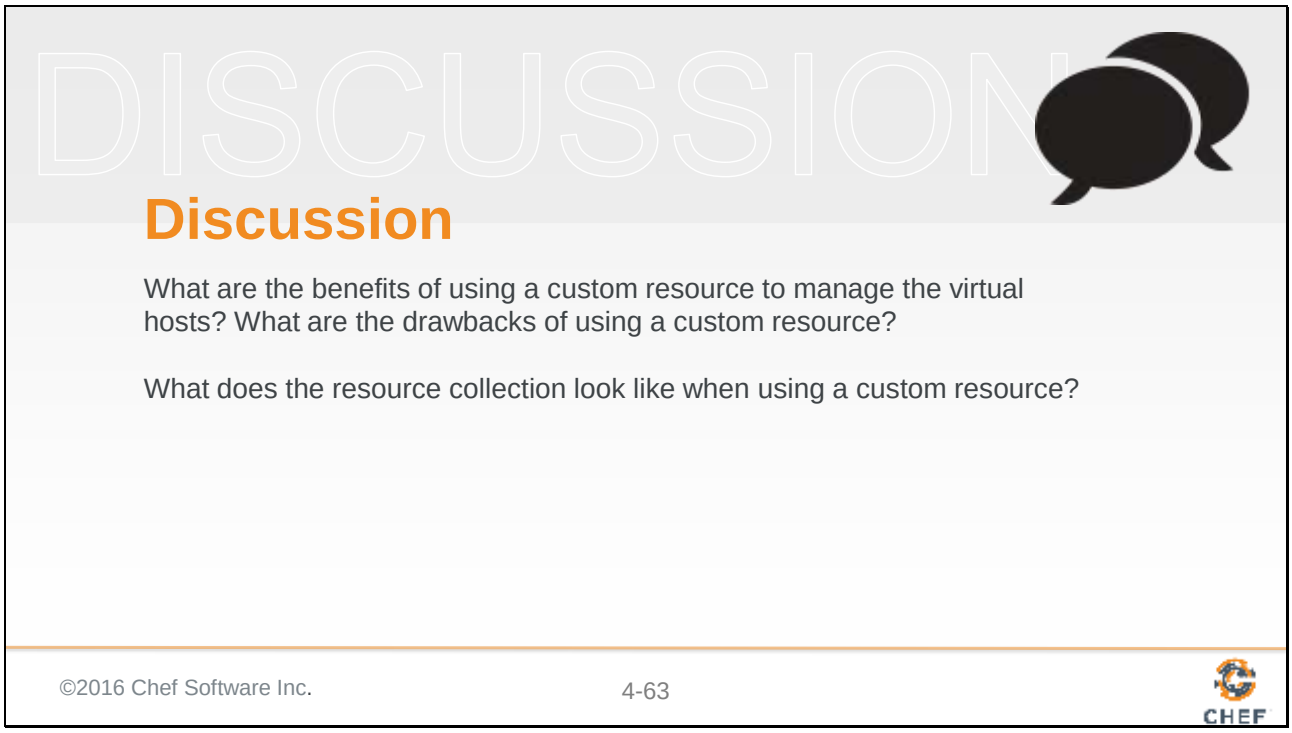
CHEF

The initial file resource has now been removed and we are creating a new apache virtual host on port 80 for our users' site. We have also updated all the expectations to correctly verify the state of the run list. Finally we also updated the tests that were executed on the virtual machine

Congratulations! The custom resource now is able to create sites and remove them. There are still more things to learn about custom resources that we will explore in the next module.

.

Slide 63



Let's finish this module with a discussion. Answer these questions. Remember that the answer "I don't know! That's why I'm here!" is a great answer.

Slide 64

What questions can we answer for you?

Slide 65