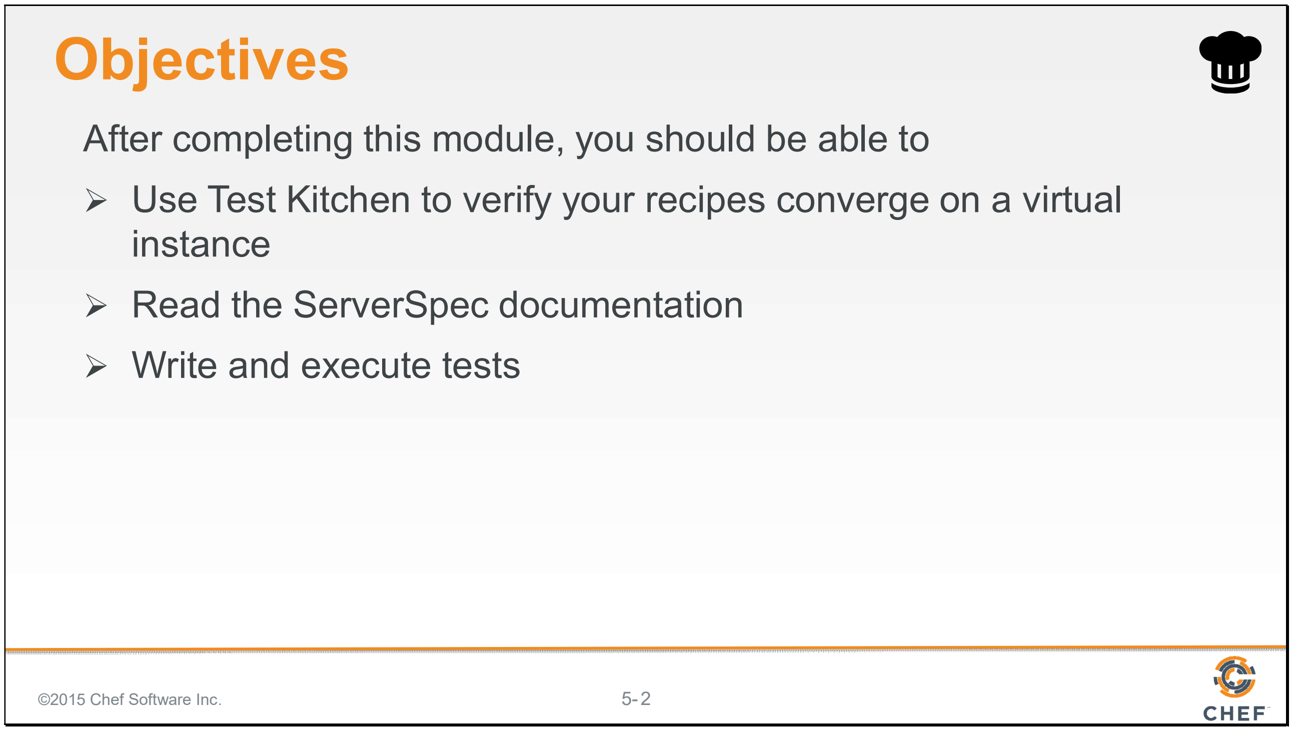
5: Testing Cookbooks

Testing Cookbooks

Validating Our Recipes in Virtual Environments

Slide 2



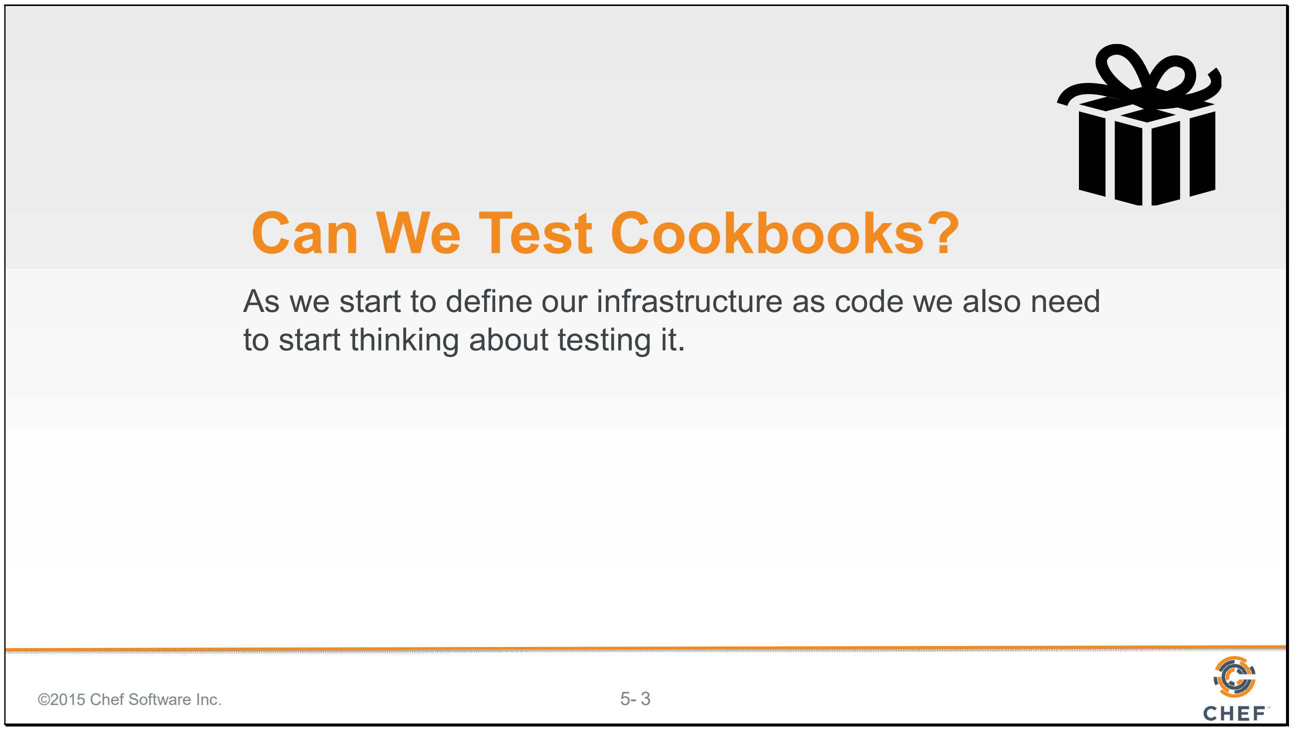
Objectives

After completing this module, you should be able to

- Use Test Kitchen to verify your recipes converge on a virtual instance
- Read the ServerSpec documentation
- Write and execute tests

In this module you will learn how to use the Test Kitchen tool to execute your configured code, write and execute tests, and use Serverspec to test your servers' actual state.

Slide 3



Can We Test Cookbooks?

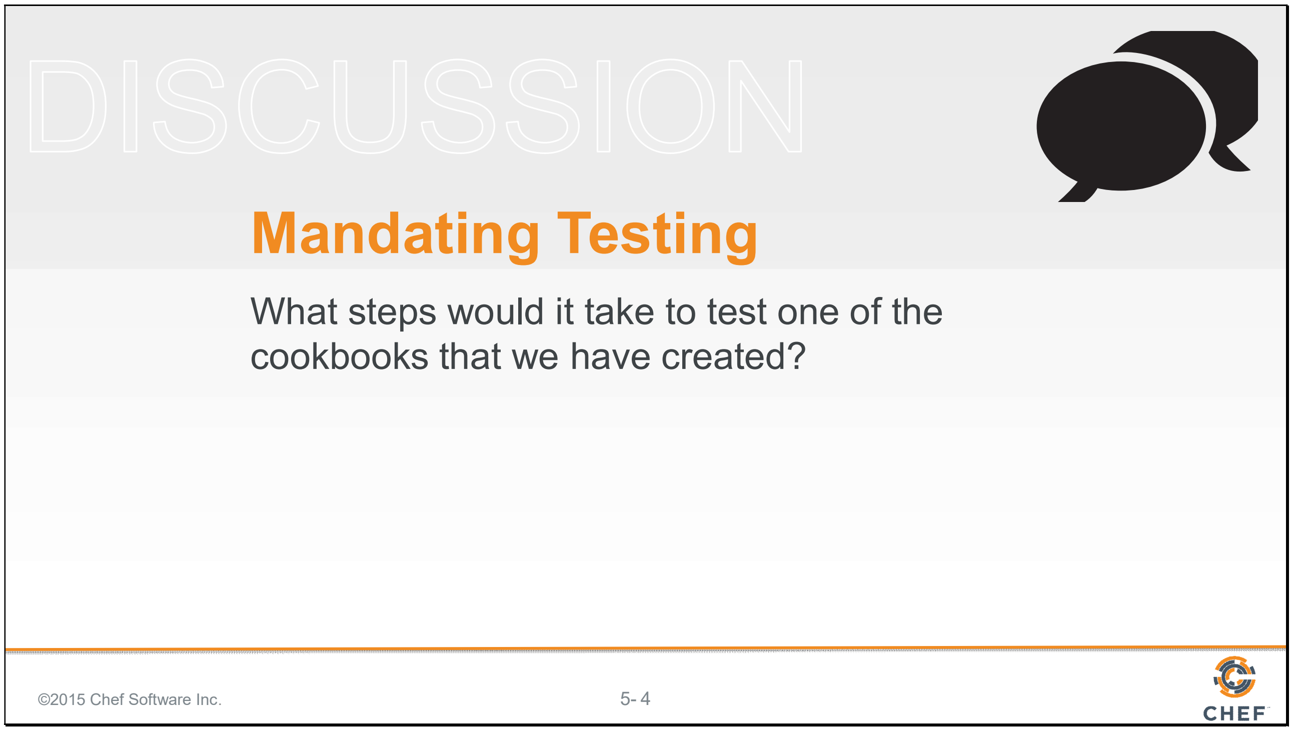
As we start to define our infrastructure as code we also need to start thinking about testing it.

©2015 Chef Software Inc. 5- 3

Will the recipes that we created work on another system similar to this one? Will they work in production?

When we develop our automation we need to start thinking about verifying it. Because it is all too common a story of automation failing when it reaches production because it was never validated against anything other than "my machine".

So how could we solve a problem like this?



DISCUSSION

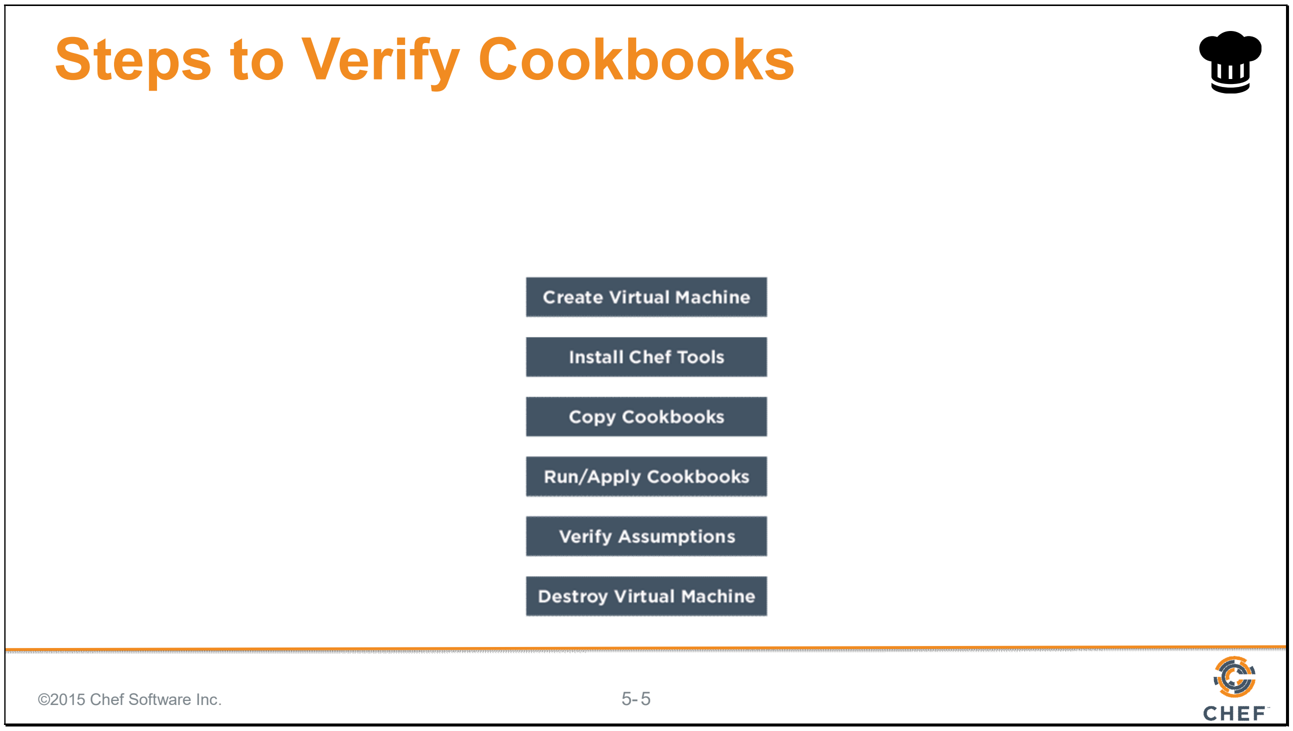
Mandating Testing

What steps would it take to test one of the cookbooks that we have created?

Write down or type out as many of the steps you can think of required to test one of the cookbooks.

When you are ready turn to another person and compare your lists. Create a complete list with all the steps that you have identified. Then as a group we will discuss all the steps necessary to test a cookbook.

Slide 5



Steps to Verify Cookbooks

Create Virtual Machine
Install Chef Tools
Copy Cookbooks
Run/Apply Cookbooks
Verify Assumptions
Destroy Virtual Machine

Here are the steps necessary to verify one of the cookbooks that you created.

Create a virtual machine or setup an instance that resembles your current production infrastructure

Install the necessary Chef tools

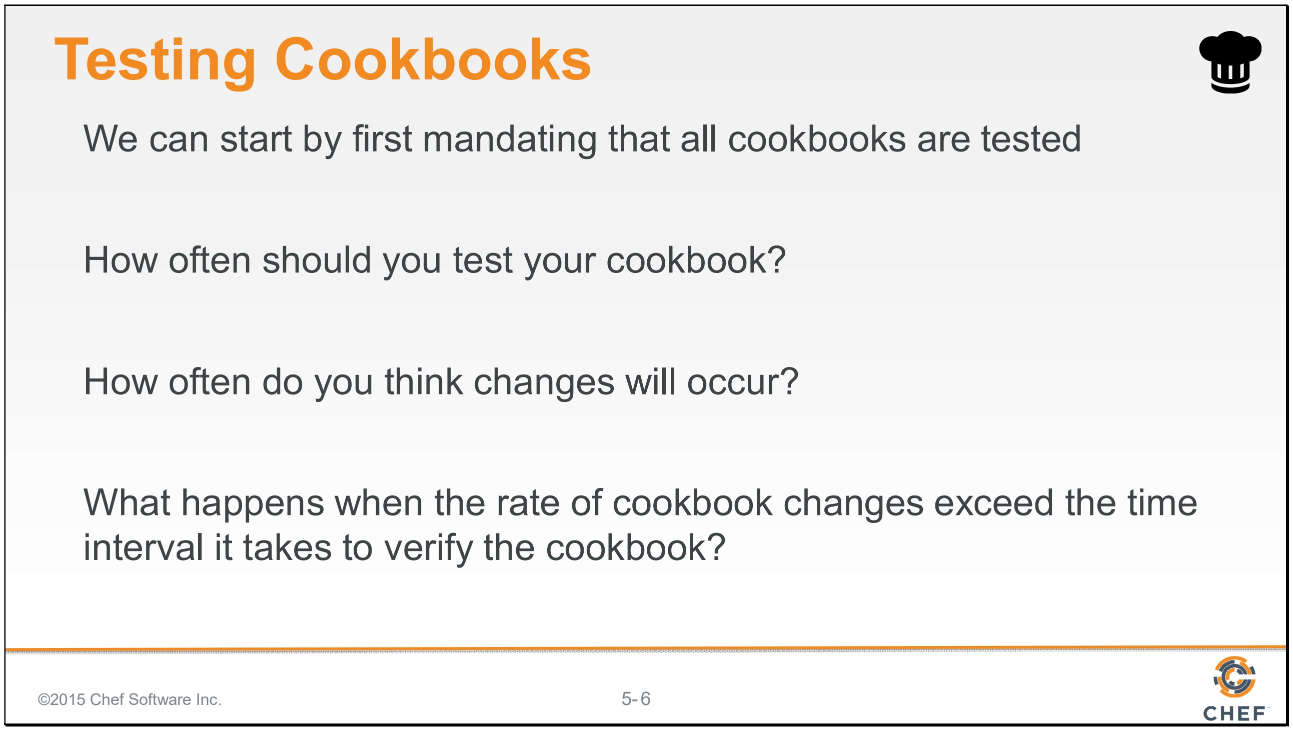
Copy the cookbooks to this new instance

Apply the cookbooks to the instance

Verify that the instance is the desired state by executing various commands

Clean up that instance by destroying it or rolling it back to a previous snapshot

Slide 6



Testing Cookbooks

We can start by first mandating that all cookbooks are tested

How often should you test your cookbook?

How often do you think changes will occur?

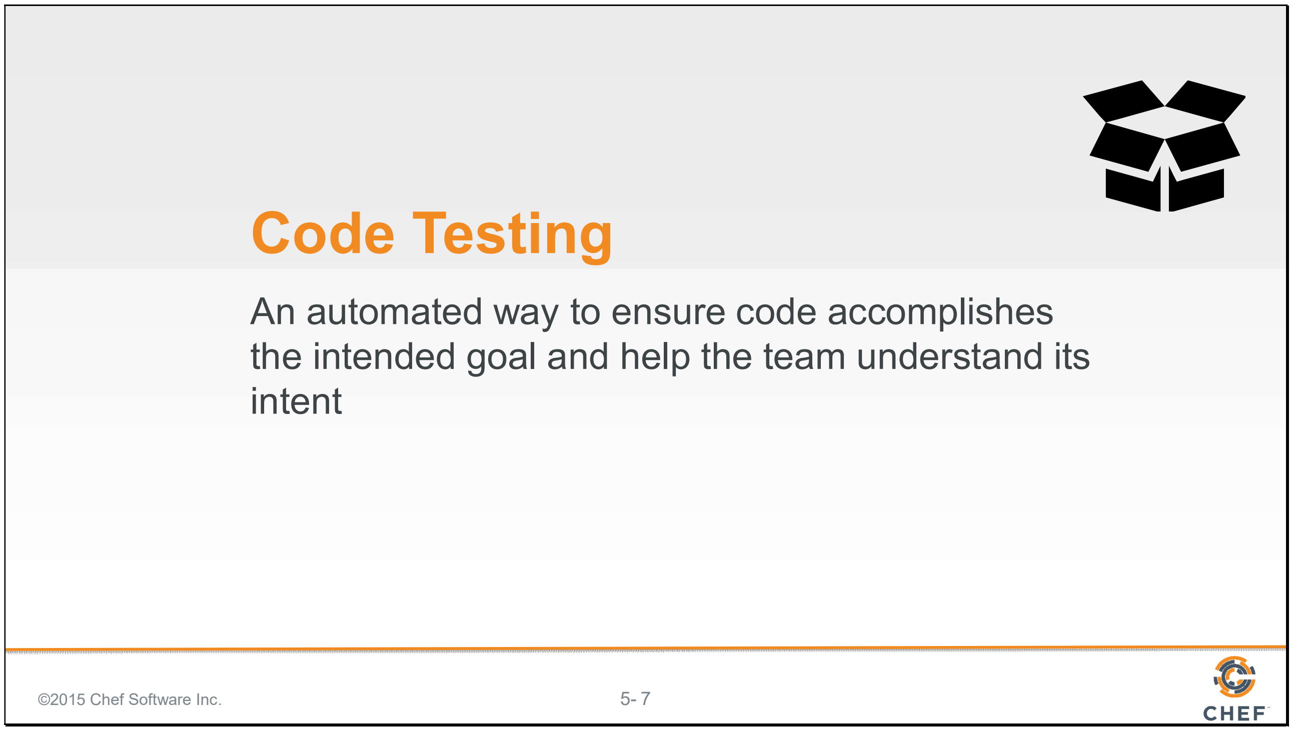
What happens when the rate of cookbook changes exceed the time interval it takes to verify the cookbook?

So we can start by mandating that all cookbooks are tested.

But we need to consider how often we need to test a cookbook and how often changes to our cookbooks will occur.

And what would happen if the rate of rate of cookbook changes exceed the time interval it takes to verify the cookbook?

Slide 7



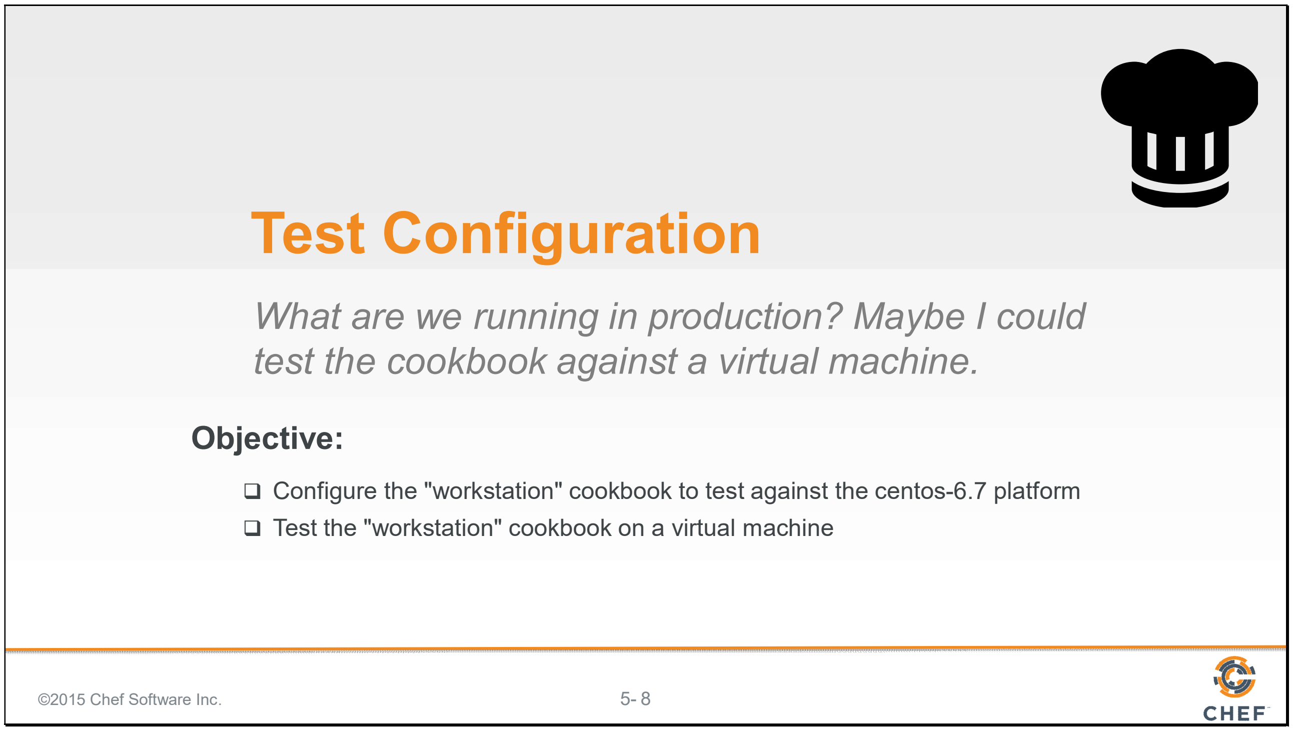
Code Testing

An automated way to ensure code accomplishes the intended goal and help the team understand its intent

Testing tools provide automated ways to ensure that the code we write accomplishes its intended goal. It also helps us understand the intent of our code by providing executable documentation. We add new cookbook features and write tests to preserve this functionality.

This provides us, or anyone else on the team, the ability to make new changes with a less likely chance of breaking something. Whether returning to the cookbook code tomorrow or in six months.

Slide 8



Test Configuration

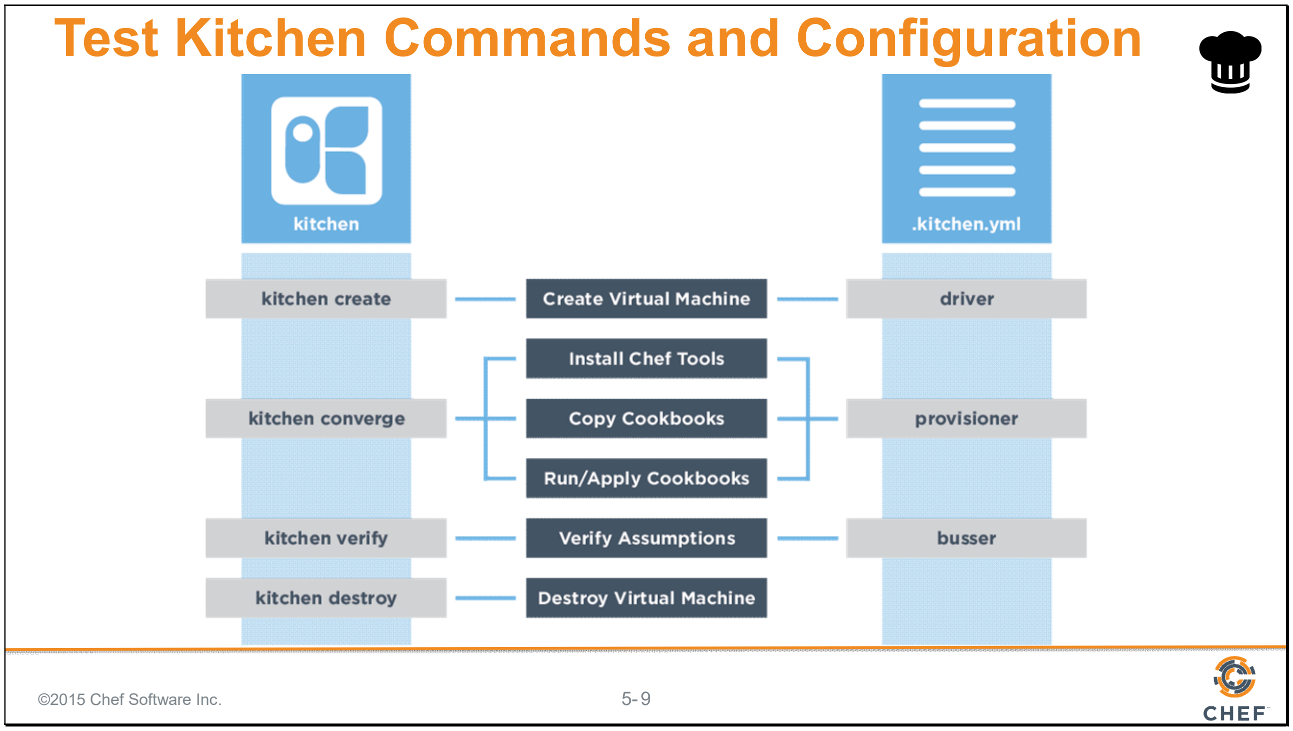
What are we running in production? Maybe I could test the cookbook against a virtual machine.

Objective:

- ❑ Configure the "workstation" cookbook to test against the centos-6.7 platform
- ❑ Test the "workstation" cookbook on a virtual machine

Well if Chef is to replace our existing tools, it is going to need to provide a way to make testing the policies more delightful.

Slide 9



Test Kitchen Commands and Configuration

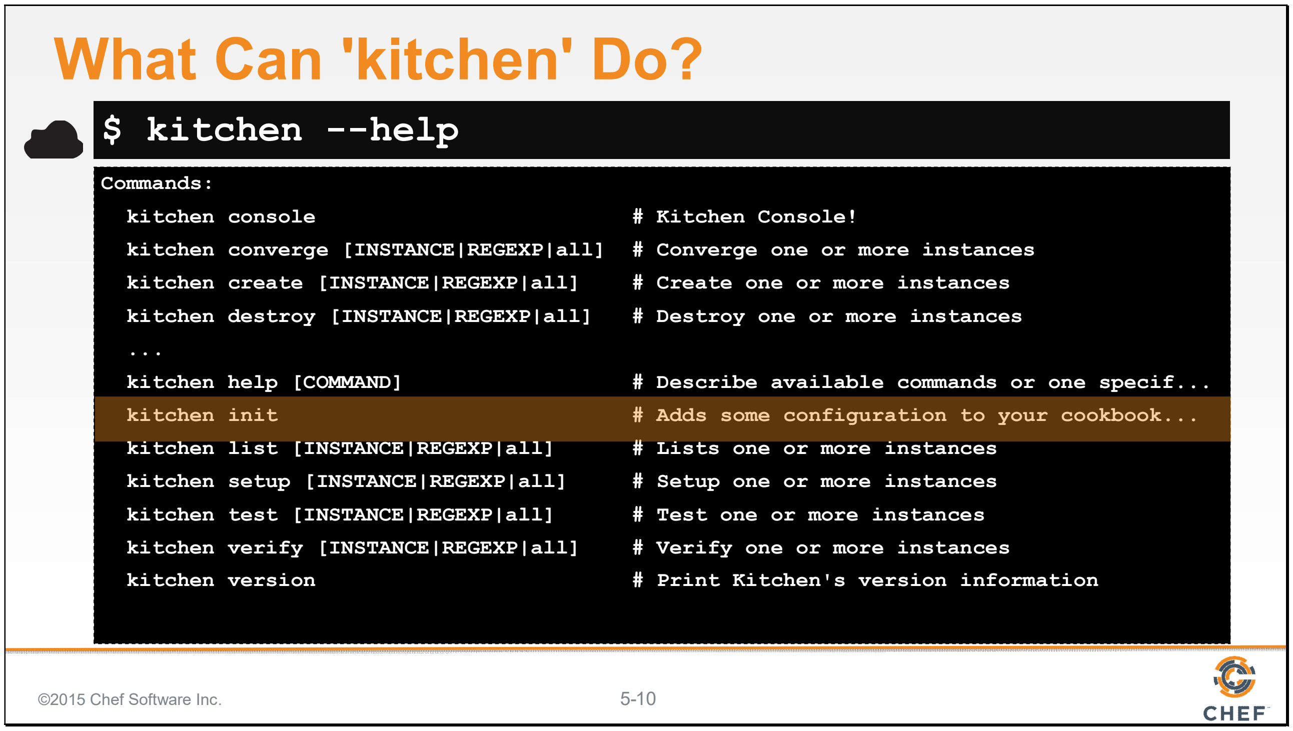
Test Kitchen allows us to create an instance solely for testing. On that created instance it will install Chef, converge a run list of recipes, verify that the instance is in the desired state, and then destroy the instance.

On the left are the kitchen commands that map to the stages of the testing lifecycle.

On the right are the kitchen configuration fields that map to the stages of the testing lifecycle.

These commands the configuration will be explained in more detail.

Slide 10



What Can 'kitchen' Do?

```
$ kitchen --help
```

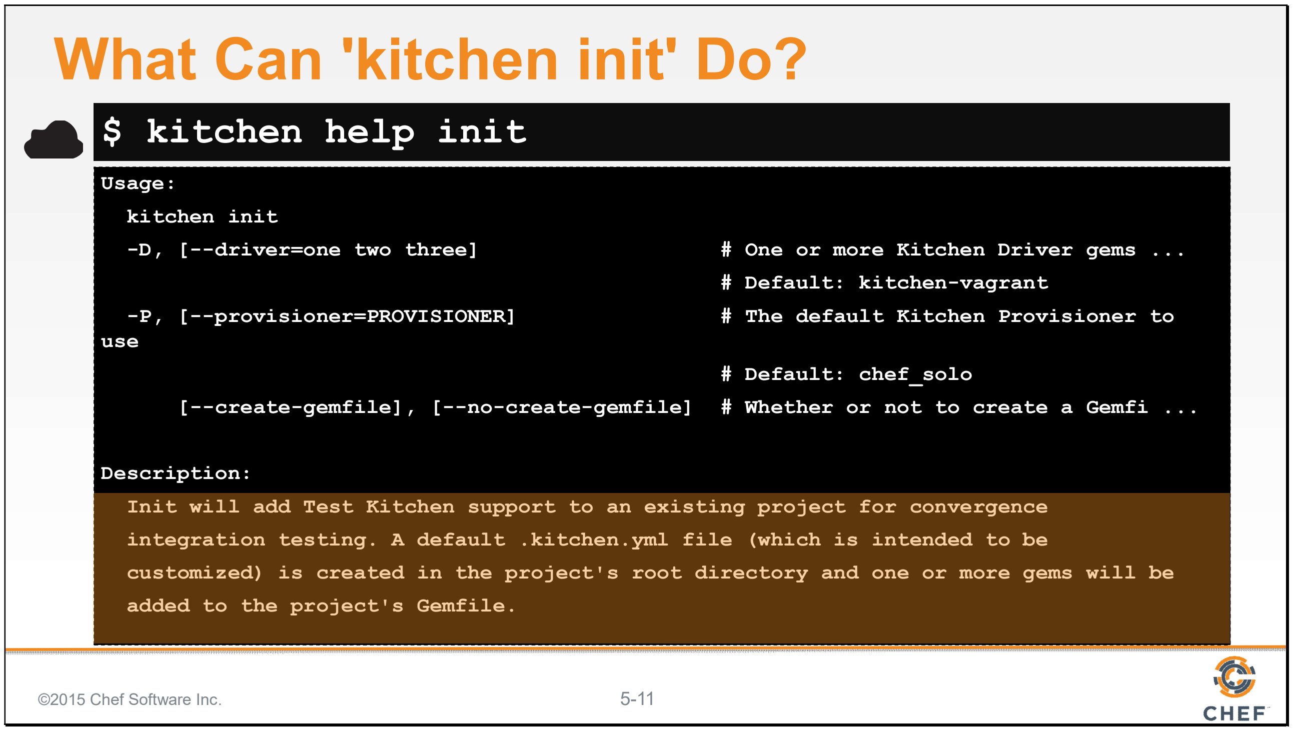
```
Commands:
  kitchen console                         # Kitchen Console!
  kitchen converge [INSTANCE|REGEXP|all]  # Converge one or more instances
  kitchen create [INSTANCE|REGEXP|all]    # Create one or more instances
  kitchen destroy [INSTANCE|REGEXP|all]   # Destroy one or more instances
  ...
  kitchen help [COMMAND]                  # Describe available commands or one specif...
  kitchen init                            # Adds some configuration to your cookbook...
  kitchen list [INSTANCE|REGEXP|all]      # Lists one or more instances
  kitchen setup [INSTANCE|REGEXP|all]     # Setup one or more instances
  kitchen test [INSTANCE|REGEXP|all]      # Test one or more instances
  kitchen verify [INSTANCE|REGEXP|all]    # Verify one or more instances
  kitchen version                         # Print Kitchen's version information
```

Kitchen is a command-line application that enables us to manage the testing lifecycle.

Similar to other tools within the ChefDK, we can ask for help to see the available commands.

The `init` command, by its name, seems like a good place to get started.

Slide 11



What Can 'kitchen init' Do?

```
$ kitchen help init
```

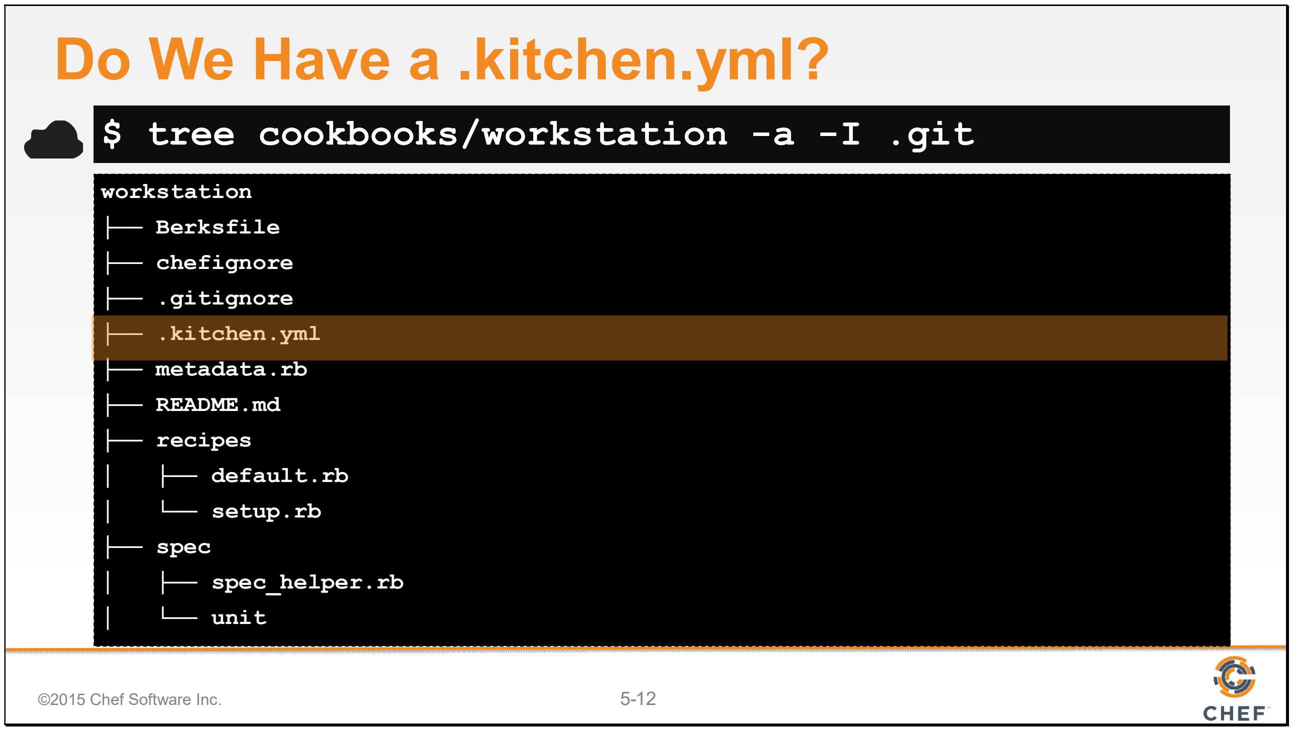
```
Usage:
  kitchen init
  -D, [--driver=one two three]                  # One or more Kitchen Driver gems ...
                                                # Default: kitchen-vagrant
  -P, [--provisioner=PROVISIONER]               # The default Kitchen Provisioner to
use
                                                # Default: chef_solo
      [--create-gemfile], [--no-create-gemfile]  # Whether or not to create a Gemfi ...

Description:
  Init will add Test Kitchen support to an existing project for convergence
  integration testing. A default .kitchen.yml file (which is intended to be
  customized) is created in the project's root directory and one or more gems will be
  added to the project's Gemfile.
```

©2015 Chef Software Inc. 5-11

`kitchen help init` tells us that it will add Test Kitchen support to an existing project. It creates a .kitchen.yml file within the project's root directory.

There are a number of flags and other options but let's see if the cookbooks we created even needs us to initialize test kitchen.

Slide 12



Do We Have a .kitchen.yml?

```
$ tree cookbooks/workstation -a -I .git
```

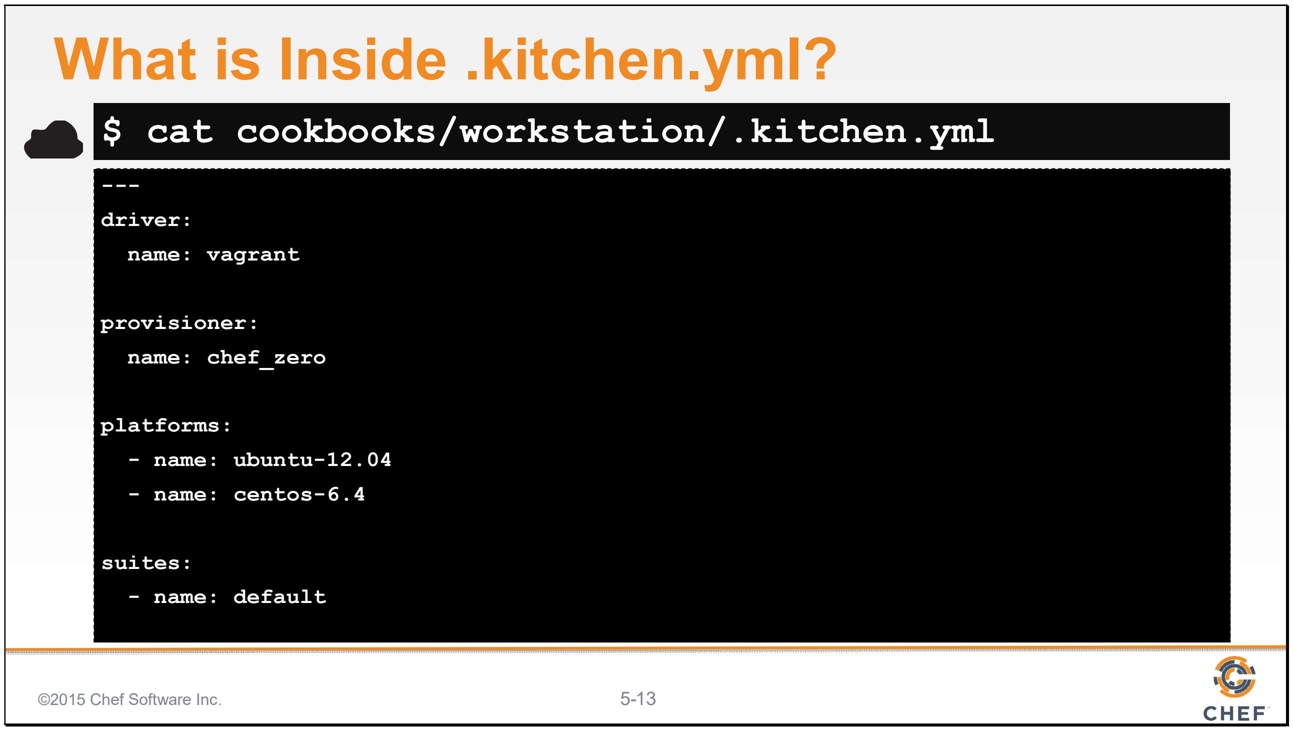
```
workstation
├── Berksfile
├── chefignore
├── .gitignore
├── .kitchen.yml
├── metadata.rb
├── README.md
├── recipes
│   ├── default.rb
│   └── setup.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
```

Using `tree` to look at the workstation cookbook, showing all hidden files and ignoring all git files, it looks like our cookbook already has a .kitchen.yml.

It was actually created alongside the other files when we ran the `chef generate cookbook` command when we originally created this cookbook.

Let's take a look at the contents of this file.

Slide 13



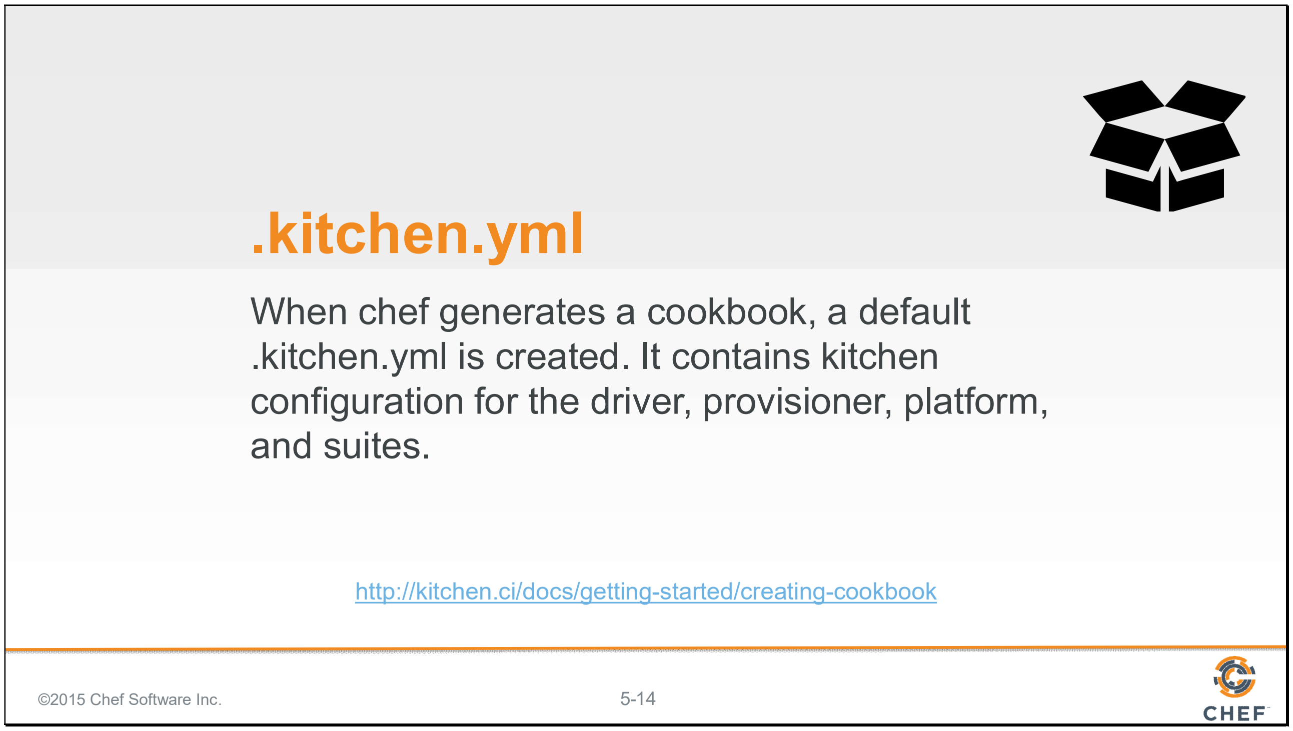
What is Inside .kitchen.yml?

```
$ cat cookbooks/workstation/.kitchen.yml
```

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-12.04
  - name: centos-6.4

suites:
  - name: default
```

The .kitchen.yml file defines a number of configuration entries that the kitchen command uses during execution.

Slide 14



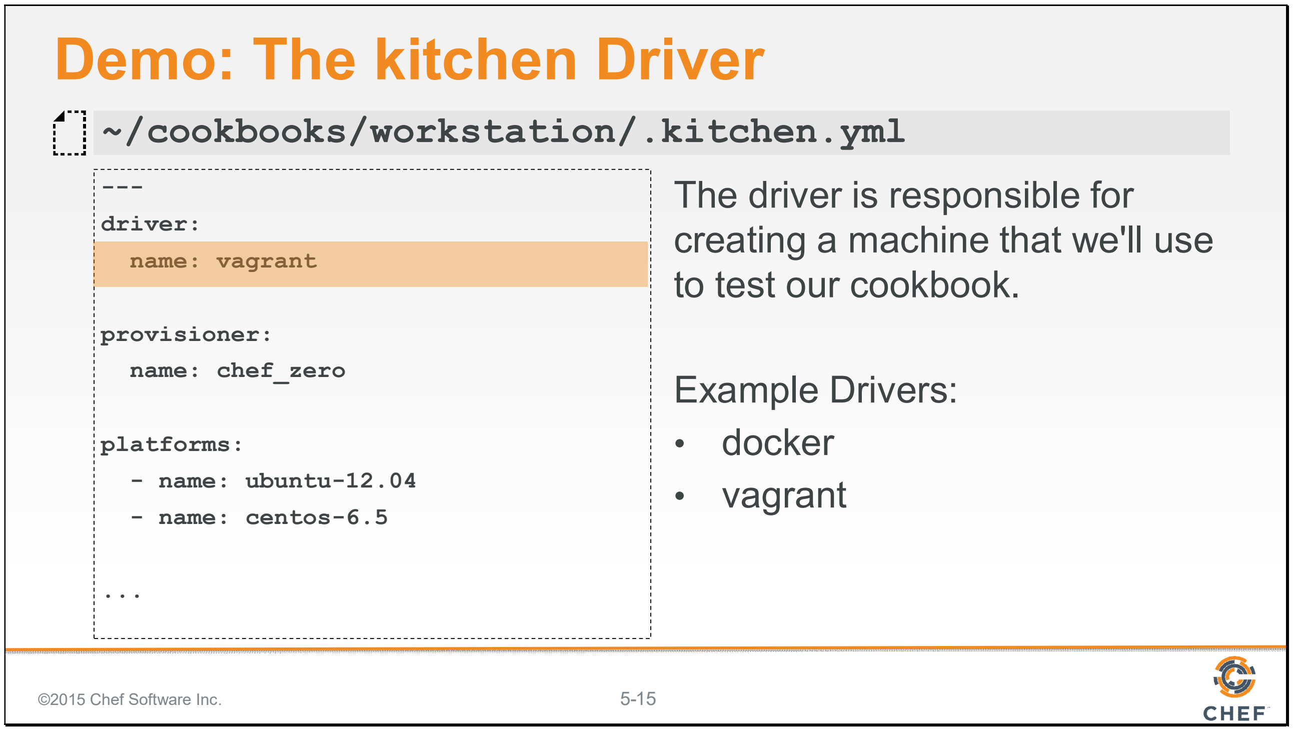
.kitchen.yml

When chef generates a cookbook, a default .kitchen.yml is created. It contains kitchen configuration for the driver, provisioner, platform, and suites.

http://kitchen.ci/docs/getting-started/creating-cookbook

We don't need to run `kitchen init` because we already have a default kitchen file. We may still need to update it to accomplish our objectives so let’s learn more about the various fields in the configuration file.

Slide 15



Demo: The kitchen Driver

~/cookbooks/workstation/.kitchen.yml

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-12.04
  - name: centos-6.5

...
```

The driver is responsible for creating a machine that we'll use to test our cookbook.

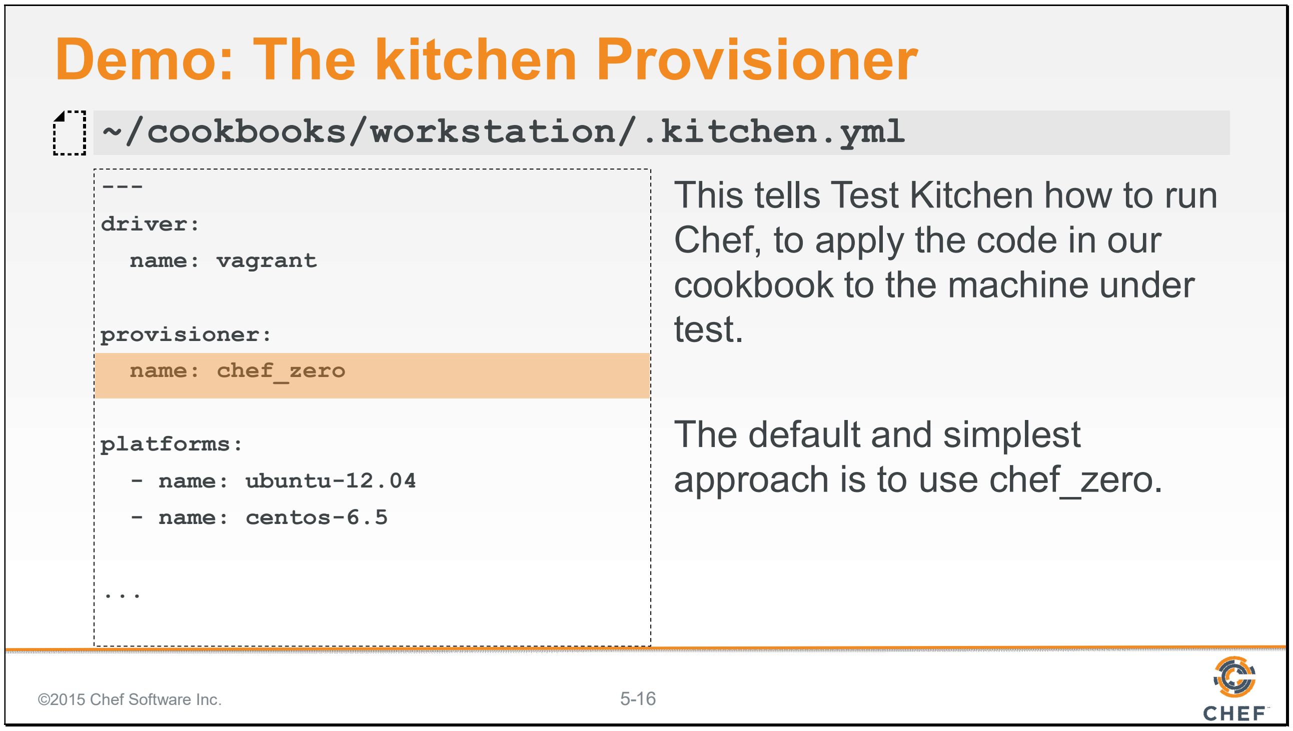
Example Drivers:

- docker
- vagrant

The first key is driver, which has a single key-value pair that specifies the name of the driver Kitchen will use when executed.

The driver is responsible for creating the instance that we will use to test our cookbook. There are lots of different drivers available--two very popular ones are the docker and vagrant driver.

Slide 16



Demo: The kitchen Provisioner

~/cookbooks/workstation/.kitchen.yml

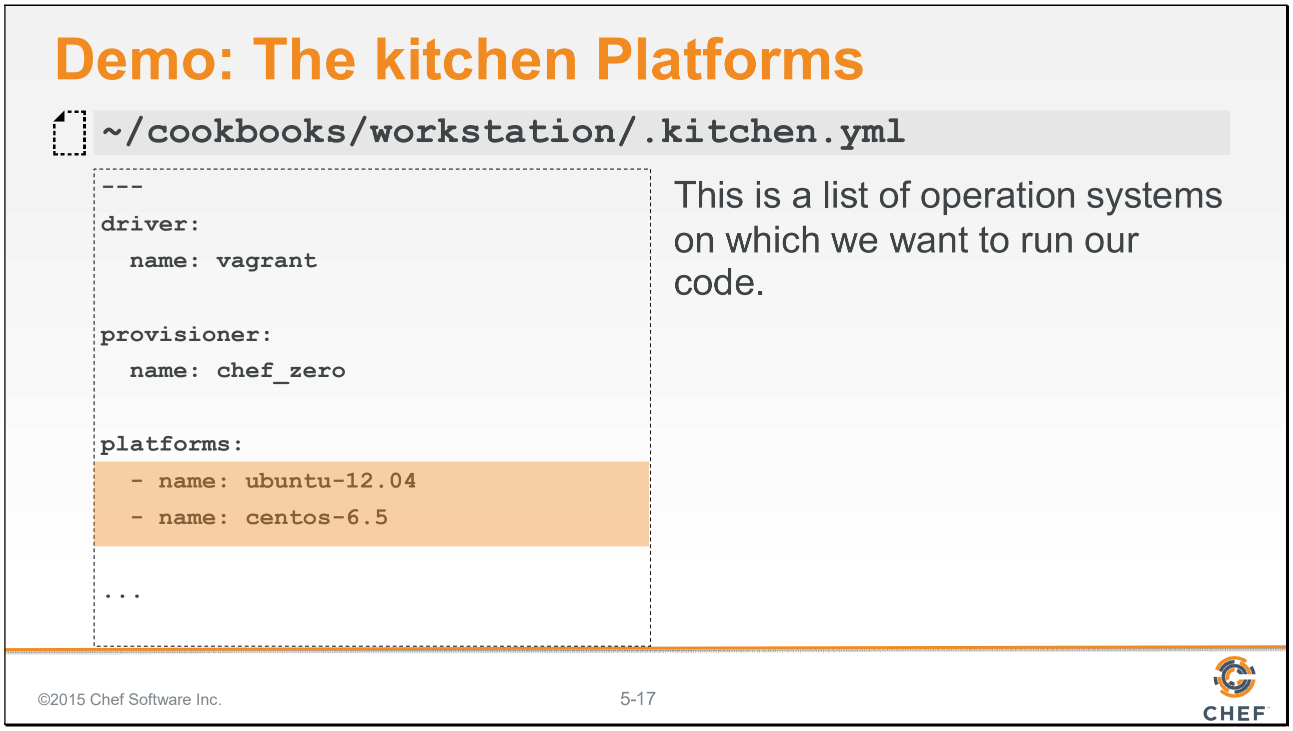
```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-12.04
  - name: centos-6.5

...
```

This tells Test Kitchen how to run Chef, to apply the code in our cookbook to the machine under test.

The default and simplest approach is to use chef_zero.

The second key is provisioner, which also has a single key-value pair which is the name of the provisioner Kitchen will use when executed. This provisioner is responsible for how it applies code to the instance that the driver created. Here the default value is chef_zero.

Slide 17



Demo: The kitchen Platforms

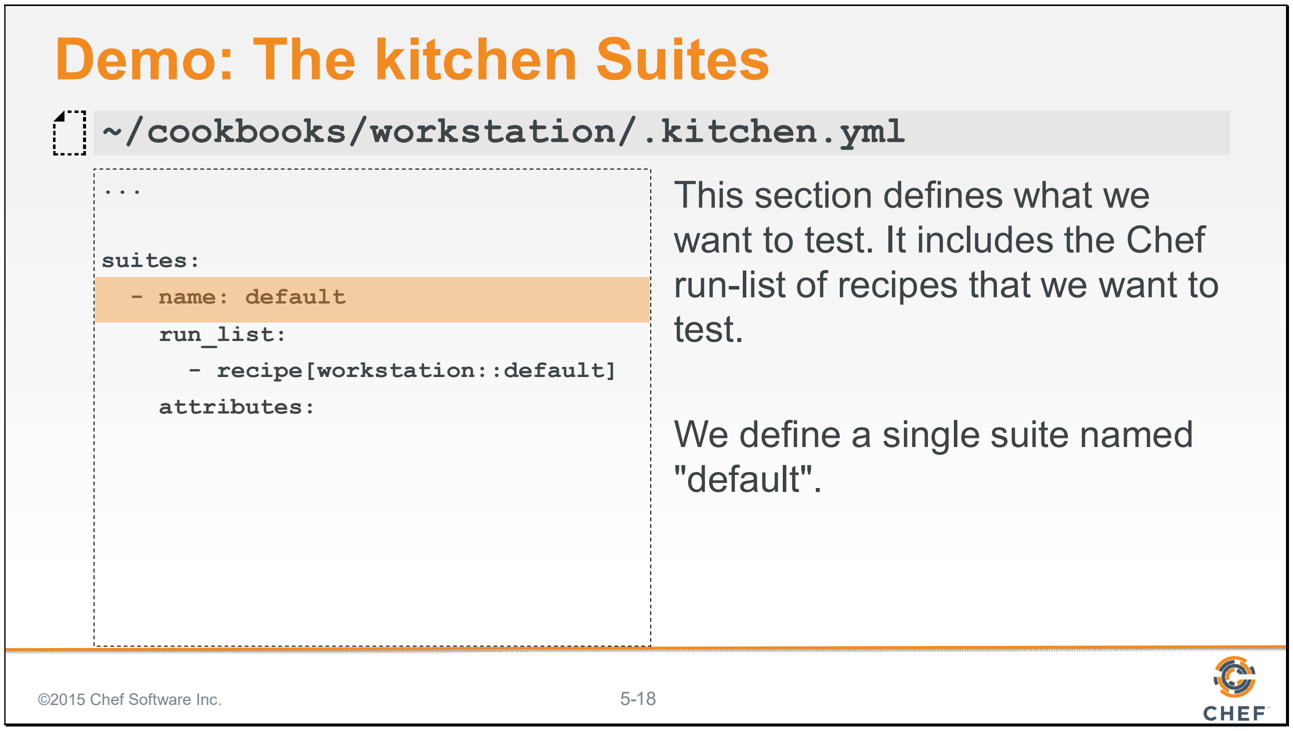
~/cookbooks/workstation/.kitchen.yml

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-12.04
  - name: centos-6.5

...
```

This is a list of operation systems on which we want to run our code.

The third key is platforms, which contains a list of all the platforms that Kitchen will test against when executed. This should be a list of all the platforms that you want your cookbook to support.

Slide 18



Demo: The kitchen Suites

~/cookbooks/workstation/.kitchen.yml

```
...

suites:
  - name: default
    run_list:
      - recipe[workstation::default]
    attributes:
```

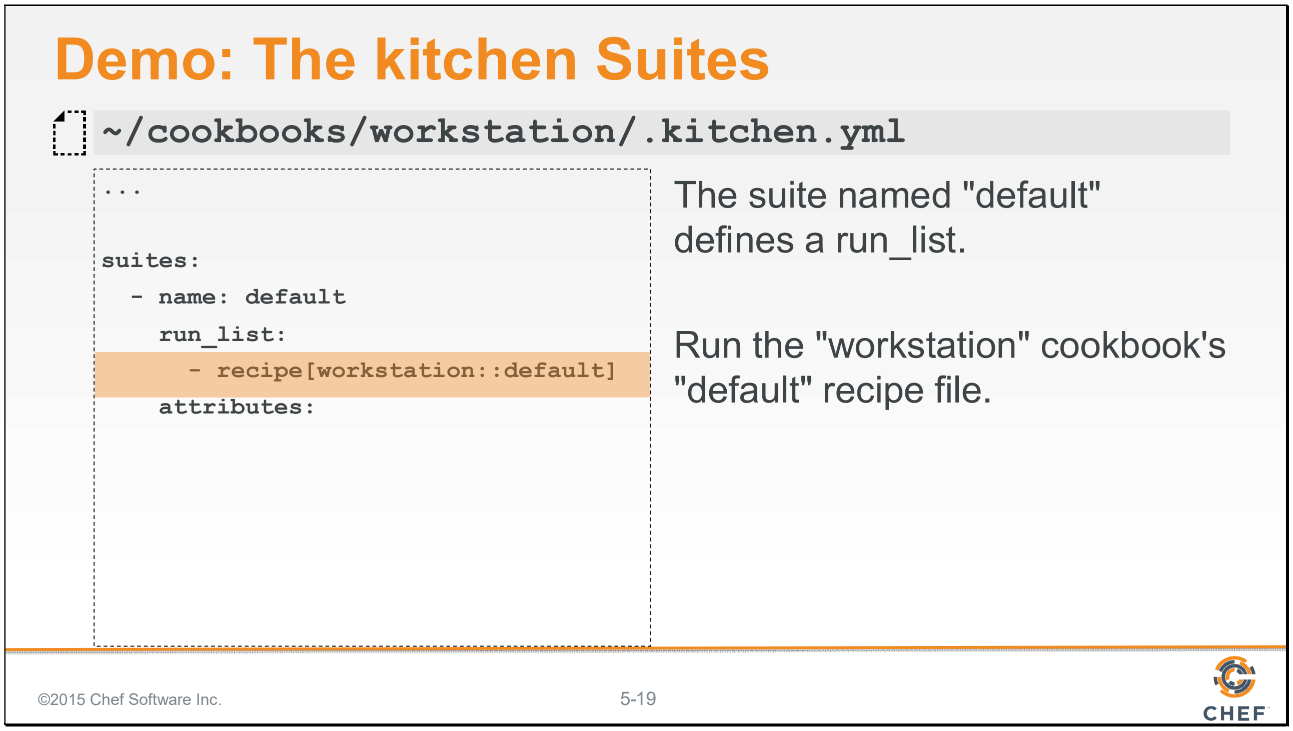
This section defines what we want to test. It includes the Chef run-list of recipes that we want to test.

We define a single suite named "default".

The fourth key is suites, which contains a list of all the test suites that Kitchen will test against when executed. Each suite usually defines a unique combination of run lists that exercise all the recipes within a cookbook.

In this example, this suite is named 'default'.

Slide 19



Demo: The kitchen Suites

~/cookbooks/workstation/.kitchen.yml

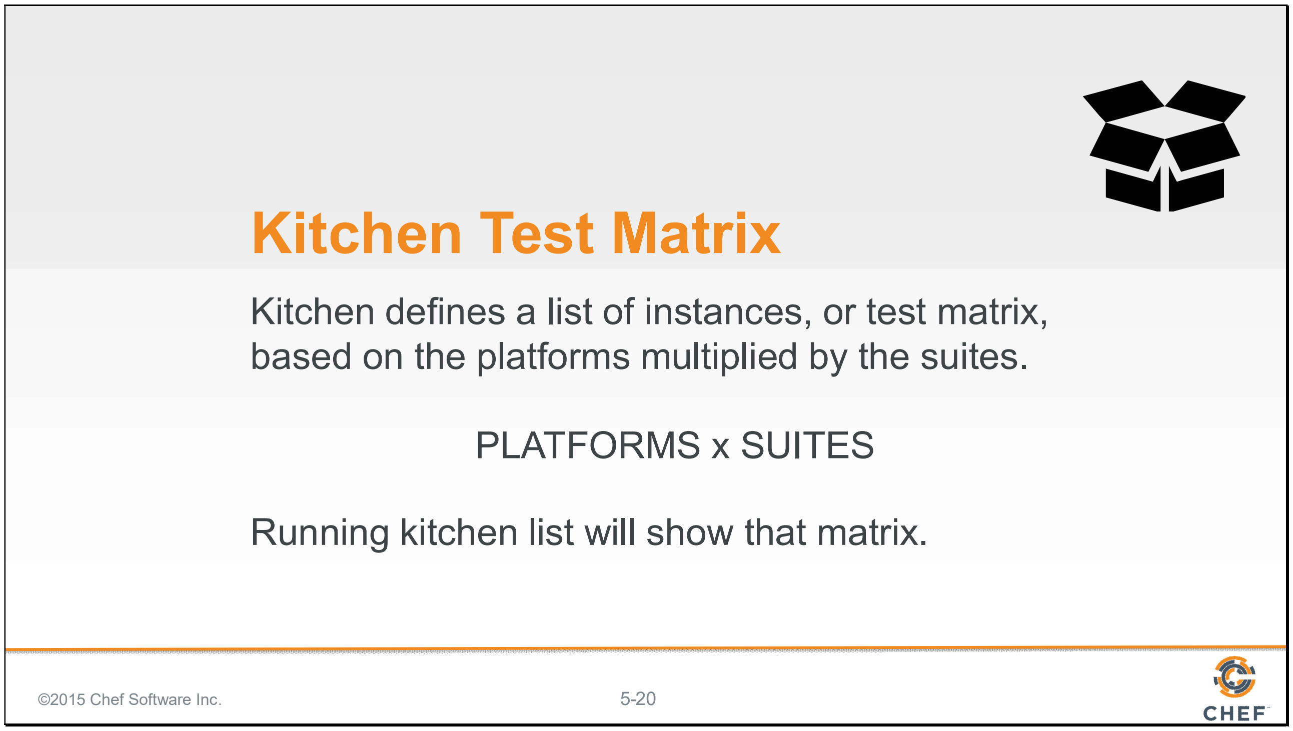
```
...

suites:
  - name: default
    run_list:
      - recipe[workstation::default]
    attributes:
```

The suite named "default" defines a run_list.

Run the "workstation" cookbook's "default" recipe file.

This default suite will execute the run list containing: The workstation cookbook's default recipe.

Slide 20



Kitchen Test Matrix

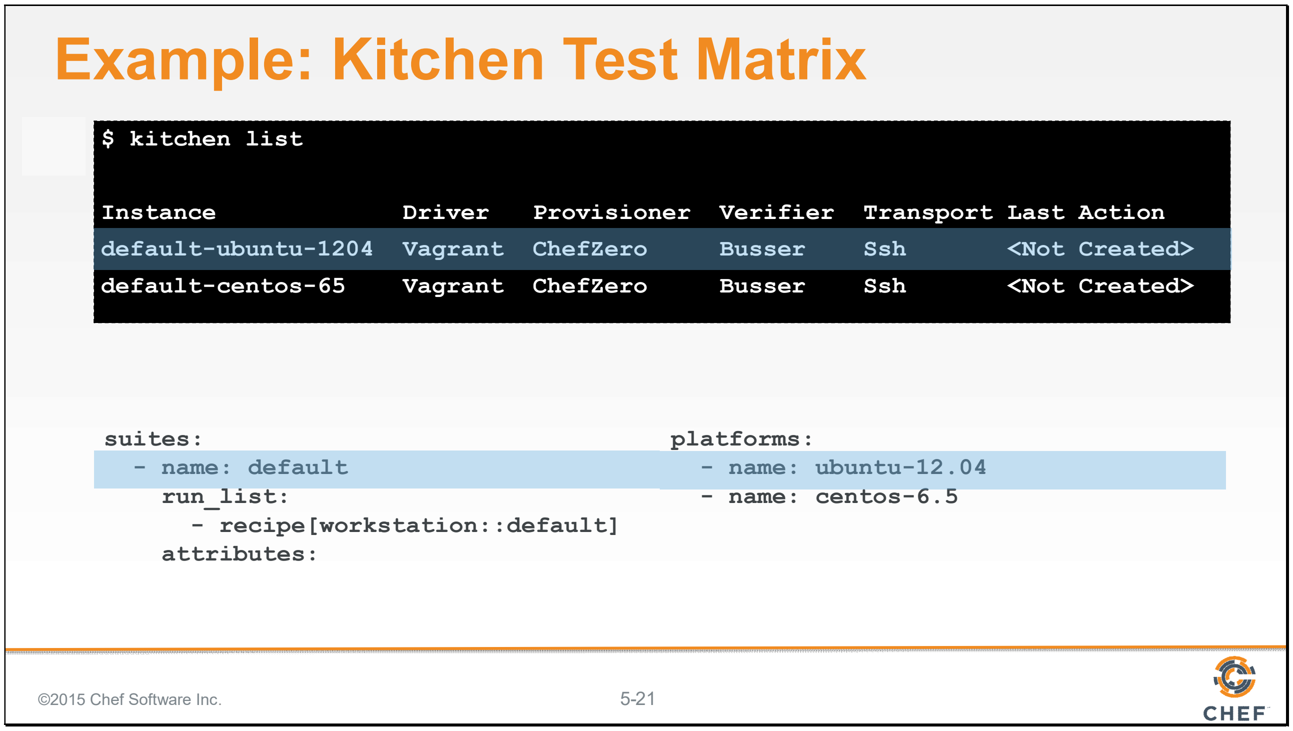
Kitchen defines a list of instances, or test matrix, based on the platforms multiplied by the suites.

PLATFORMS x SUITES

Running kitchen list will show that matrix.

It is important to recognize that within the .kitchen.yml file we defined two fields that create a test matrix; the number of platforms we want to support multiplied by the number of test suites that we defined.

Slide 21



Example: Kitchen Test Matrix

```
$ kitchen list

Instance              Driver   Provisioner  Verifier  Transport Last Action
default-ubuntu-1204   Vagrant  ChefZero     Busser    Ssh       <Not Created>
default-centos-65     Vagrant  ChefZero     Busser    Ssh       <Not Created>
```

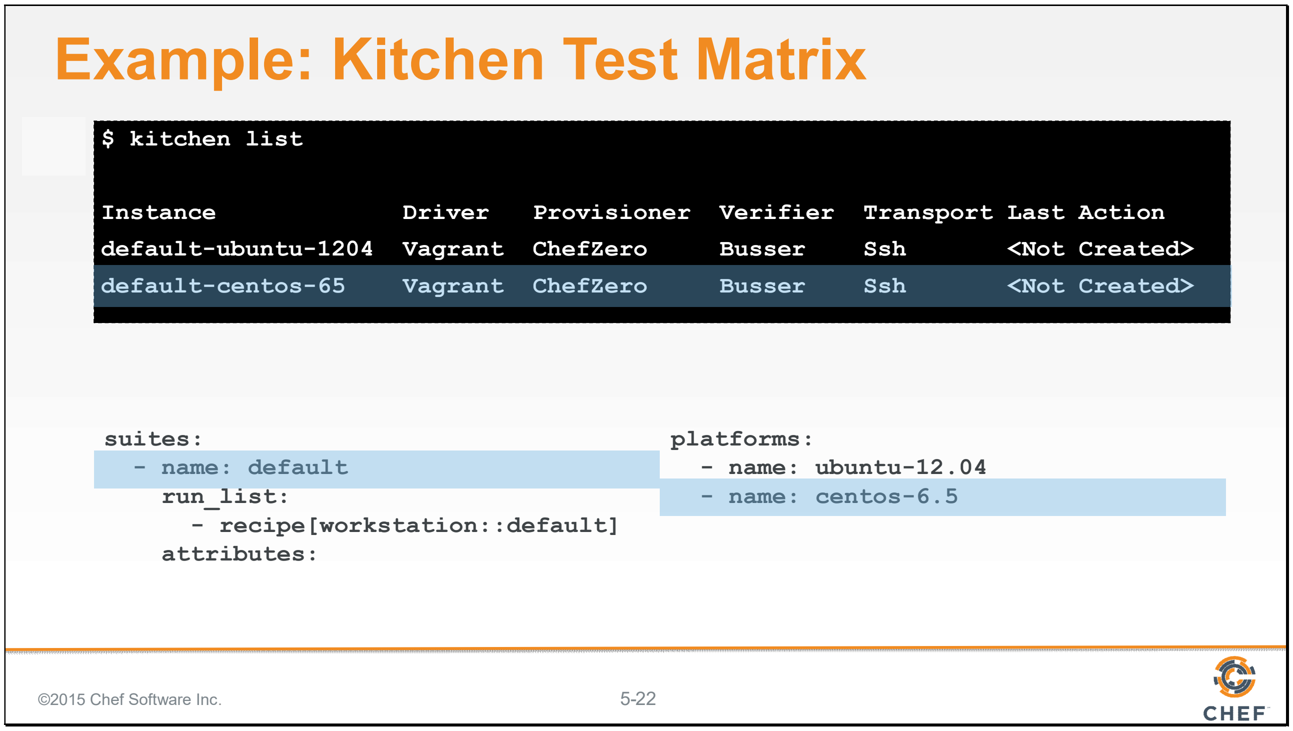
```
suites:                                        platforms:
  - name: default                                - name: ubuntu-12.04
    run_list:                                    - name: centos-6.5
      - recipe[workstation::default]
    attributes:
```

©2015 Chef Software Inc. 5-21

We can visualize this test matrix by running the command `kitchen list`.

In the output you can see that an instance is created in the list for every suite and every platform. In our current file we have one suite, named 'default', and two platforms. First the ubuntu 12.04 platform.

Slide 22



Example: Kitchen Test Matrix

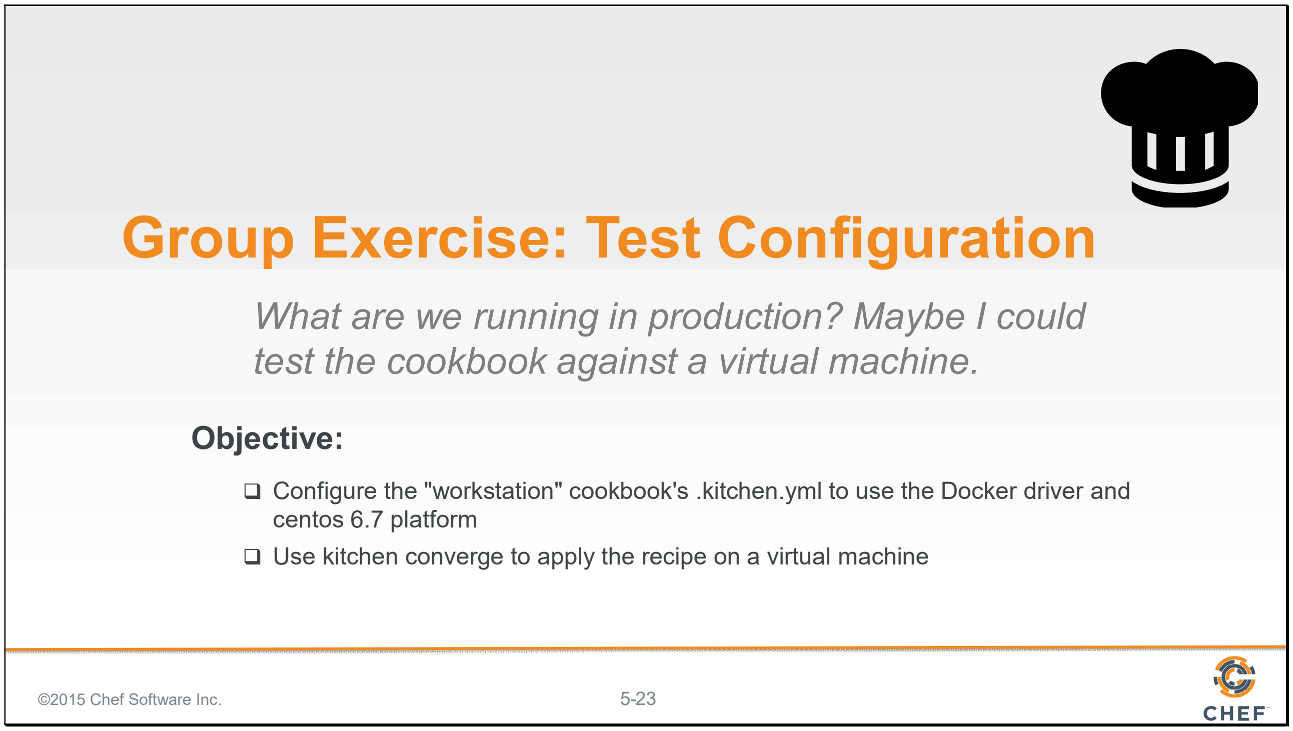
```
$ kitchen list

Instance              Driver   Provisioner  Verifier  Transport Last Action
default-ubuntu-1204   Vagrant  ChefZero     Busser    Ssh       <Not Created>
default-centos-65     Vagrant  ChefZero     Busser    Ssh       <Not Created>
```

```
suites:
  - name: default
    run_list:
      - recipe[workstation::default]
    attributes:
```

```
platforms:
  - name: ubuntu-12.04
  - name: centos-6.5
```

We can visualize this test matrix by running the command `kitchen list`.

In the output you can see that an instance is created in the list for every suite and every platform. In our current file we have one suite, named 'default', and two platforms. First the ubuntu 12.04 platform.



Group Exercise: Test Configuration

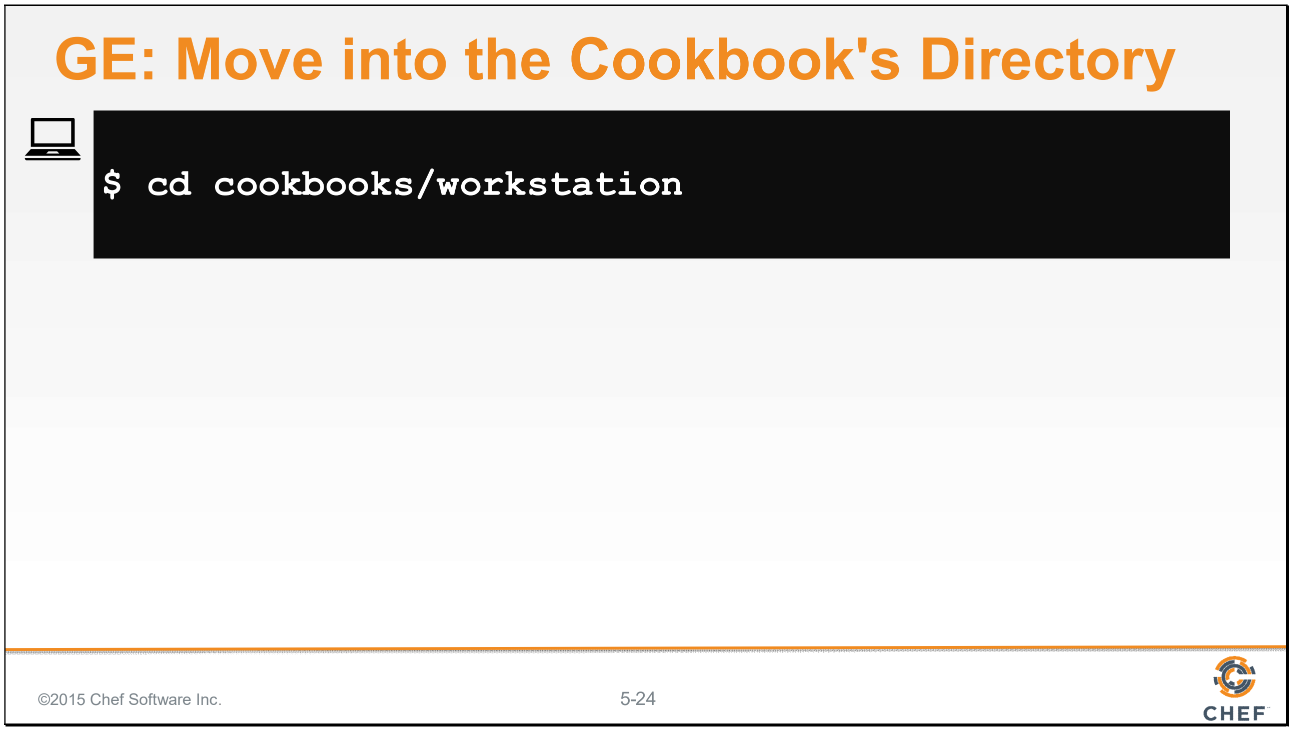
What are we running in production? Maybe I could test the cookbook against a virtual machine.

Objective:

- ❑ Configure the "workstation" cookbook's .kitchen.yml to use the Docker driver and centos 6.7 platform
- ❑ Use kitchen converge to apply the recipe on a virtual machine

Remembering our objective, we want to update our .kitchen.yml file to use the Docker driver and we want to test against a single platform named centos 6.7.

Slide 24

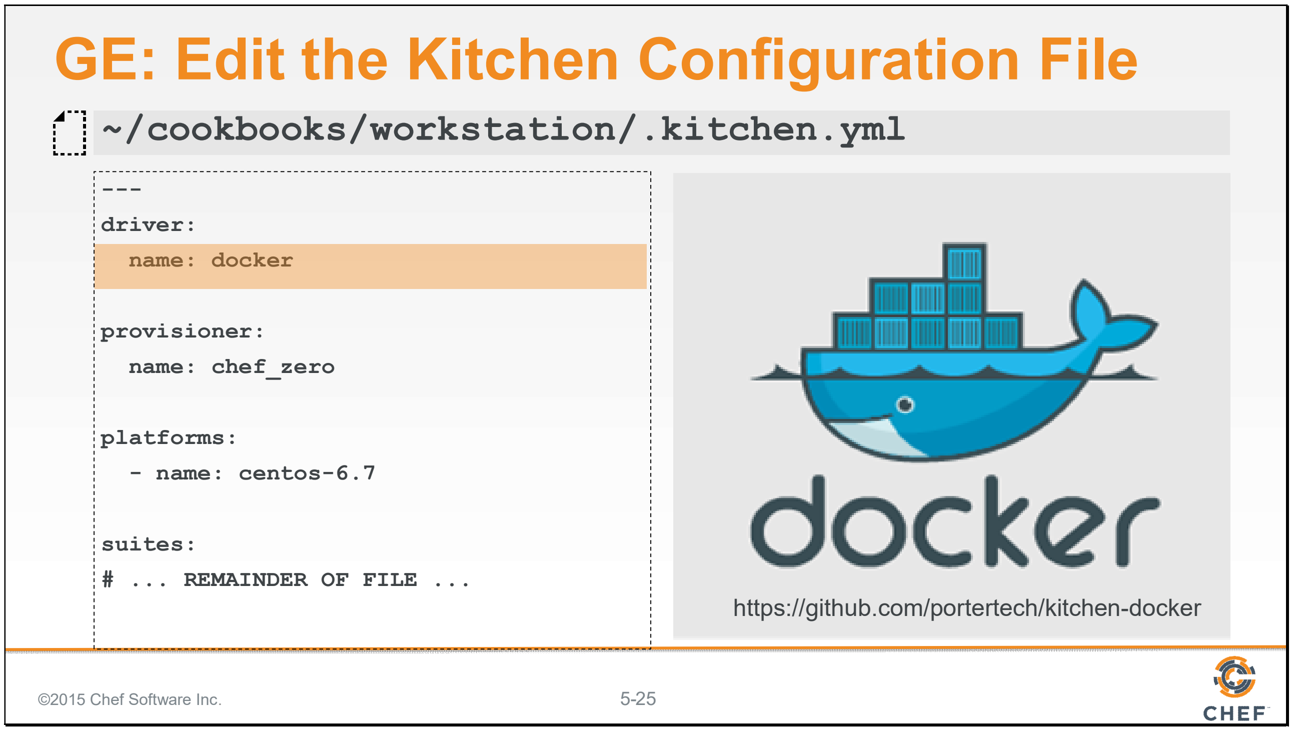


GE: Move into the Cookbook's Directory

```
$ cd cookbooks/workstation
```

Let's change into our workstation cookbook's directory.

Slide 25



GE: Edit the Kitchen Configuration File

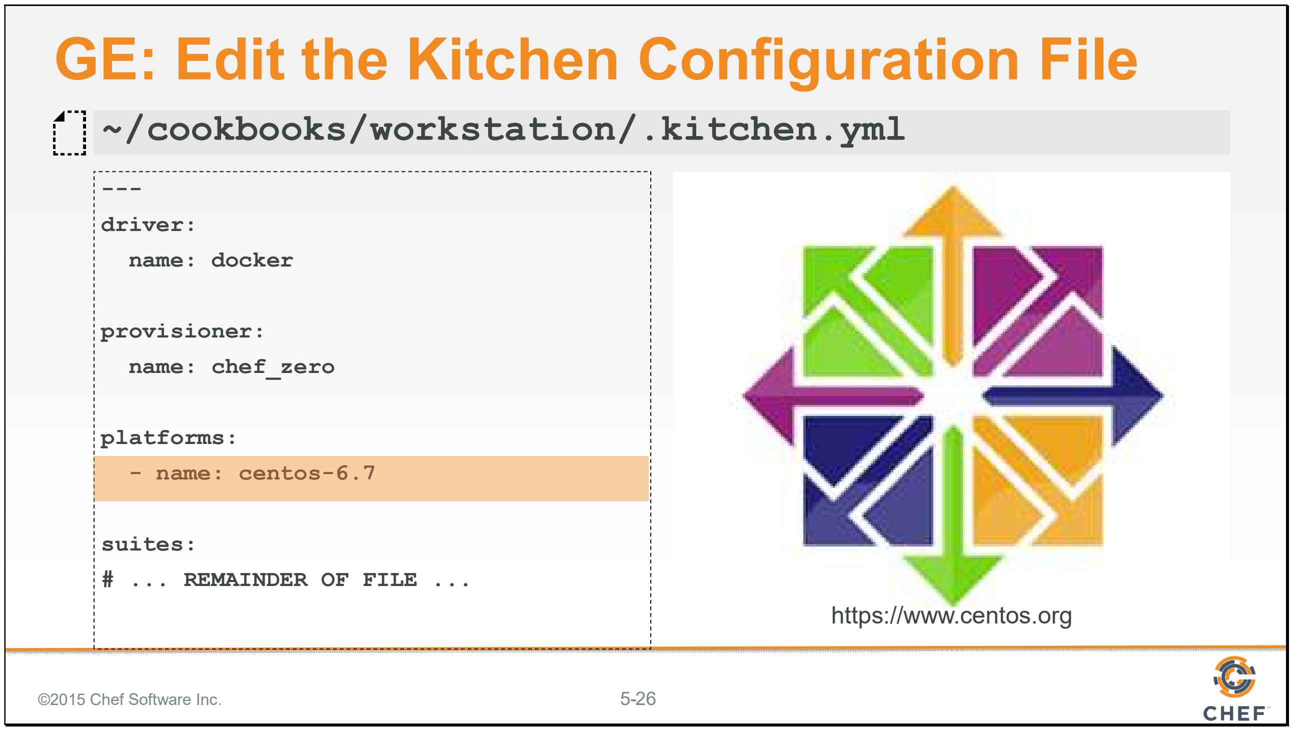
~/cookbooks/workstation/.kitchen.yml

```
---
driver:
  name: docker

provisioner:
  name: chef_zero

platforms:
  - name: centos-6.7

suites:
# ... REMAINDER OF FILE ...
```

https://github.com/portertech/kitchen-docker

Docker is a driver. So replace the existing vagrant driver, in your .kitchen.yml, with the Docker driver.

Slide 26



GE: Edit the Kitchen Configuration File

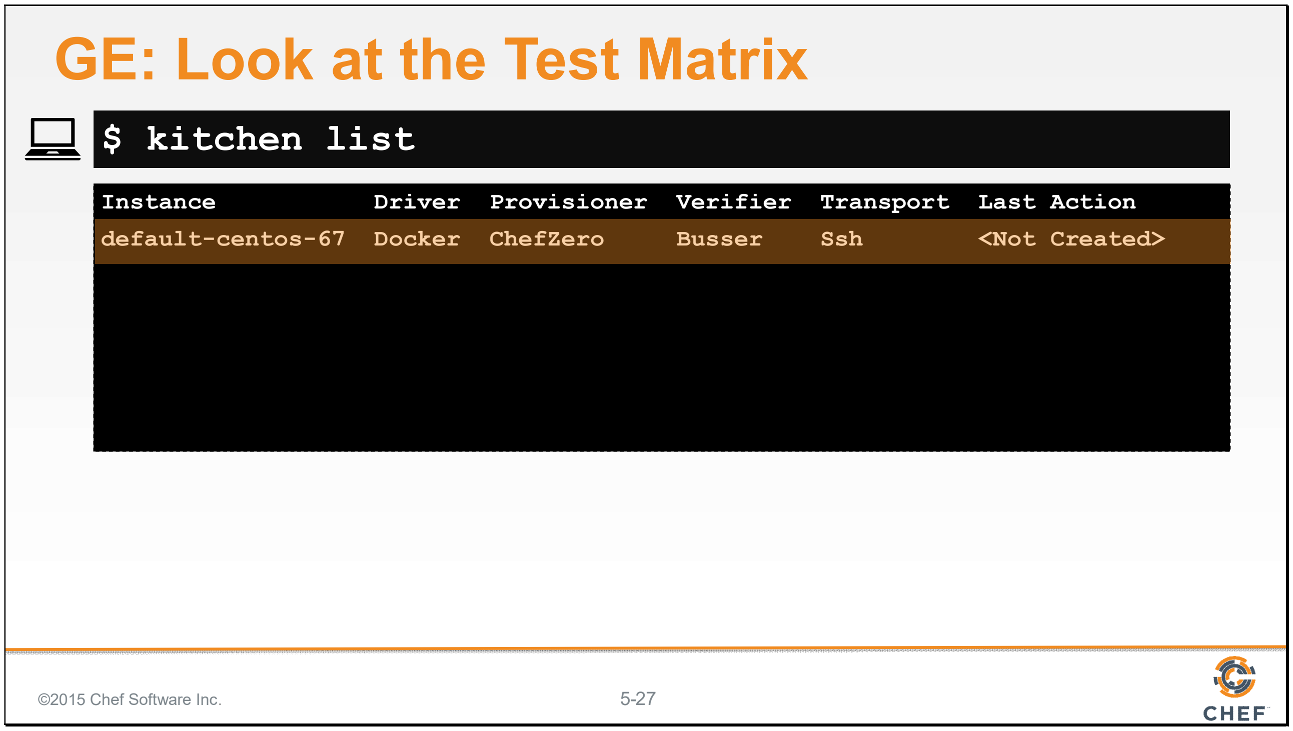
~/cookbooks/workstation/.kitchen.yml

```
---
driver:
  name: docker

provisioner:
  name: chef_zero

platforms:
  - name: centos-6.7

suites:
# ... REMAINDER OF FILE ...
```

https://www.centos.org

We also want to update our platforms to list only centos-6.7.

Slide 27



GE: Look at the Test Matrix

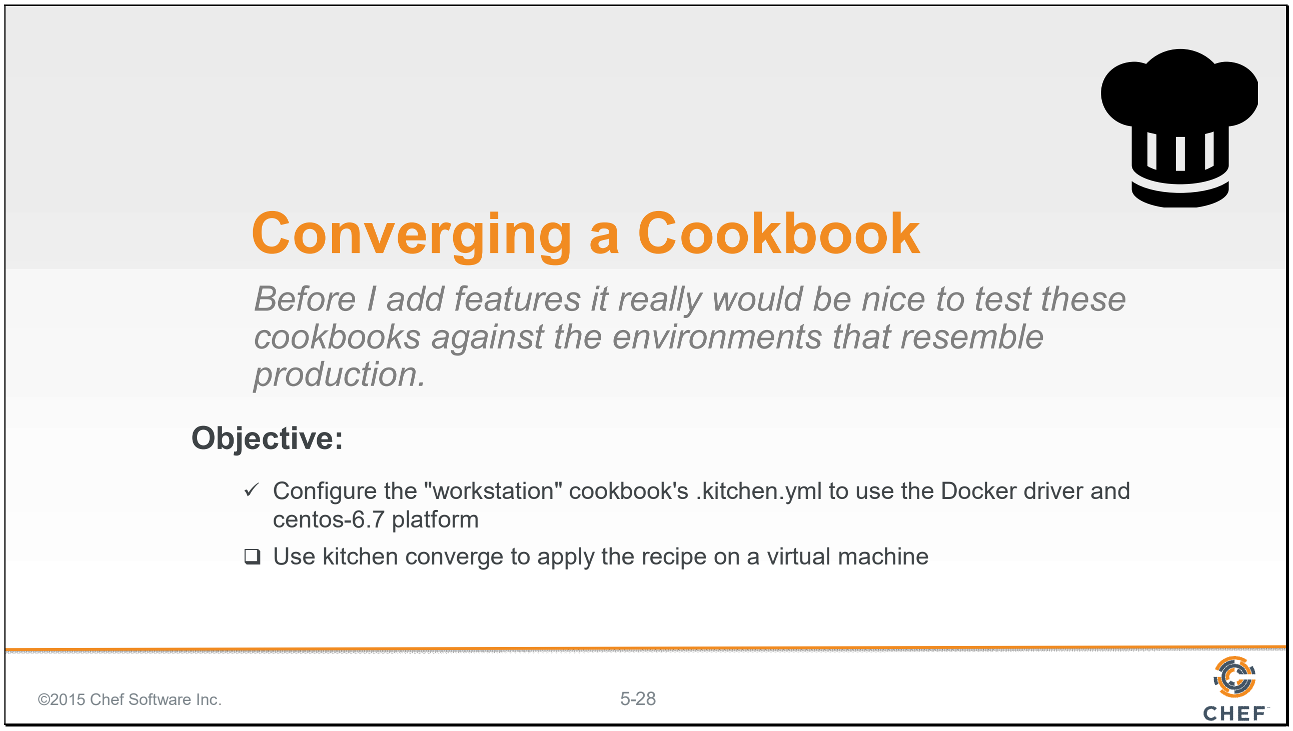
```
$ kitchen list
```

```
Instance           Driver  Provisioner  Verifier  Transport  Last Action
default-centos-67  Docker  ChefZero     Busser    Ssh        <Not Created>
```

CHEF

Run the `kitchen list` command to display our test matrix. You should see a single instance.

Slide 28



Converging a Cookbook

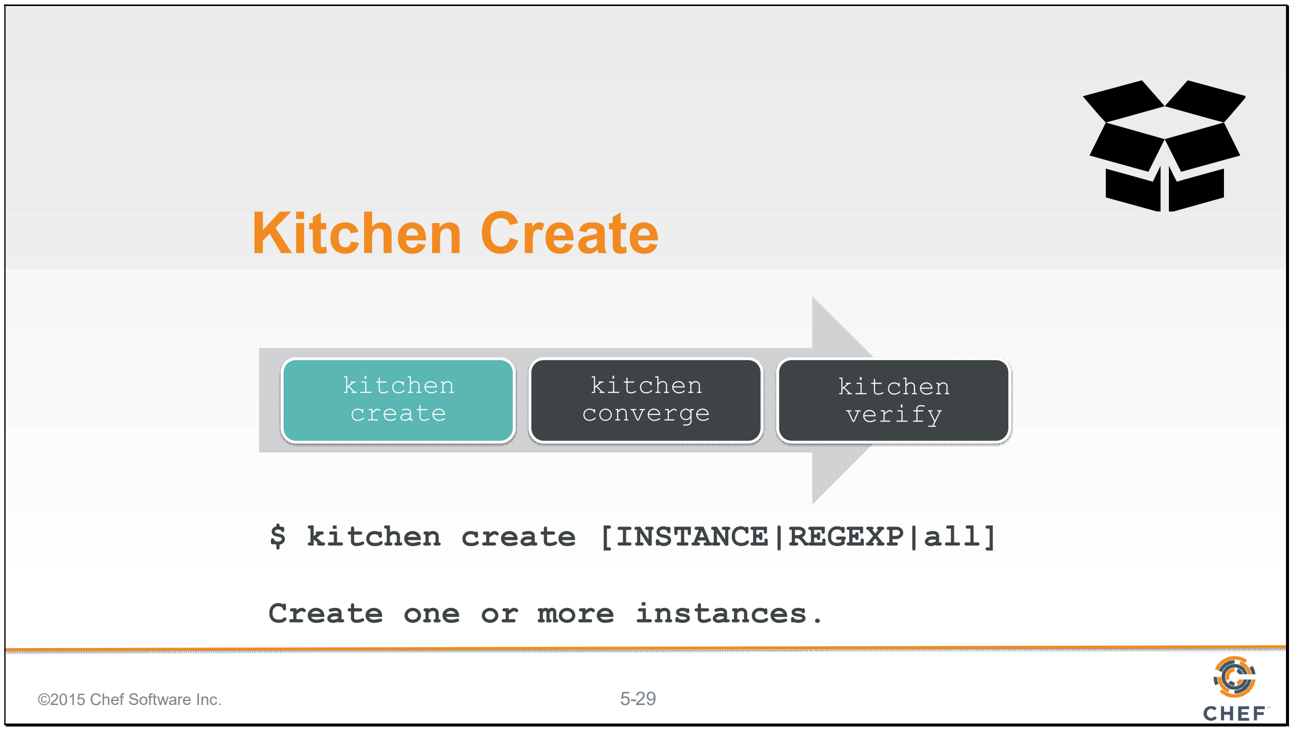
Before I add features it really would be nice to test these cookbooks against the environments that resemble production.

Objective:

- ✓ Configure the "workstation" cookbook's .kitchen.yml to use the Docker driver and centos-6.7 platform
- ❑ Use kitchen converge to apply the recipe on a virtual machine

Now that we've defined the test matrix that we want to support, it is time to understand how to use Test Kitchen to create an instance, converge a run list of recipes on that instance, verify that the instance is in the desired state, and then destroy the instance.

Slide 29



Kitchen Create

kitchen create
kitchen converge
kitchen verify

```
$ kitchen create [INSTANCE|REGEXP|all]
```

Create one or more instances.

The first kitchen command is `kitchen create`.

To create an instance means to turn on virtual or cloud instances for the platforms specified in the kitchen configuration.

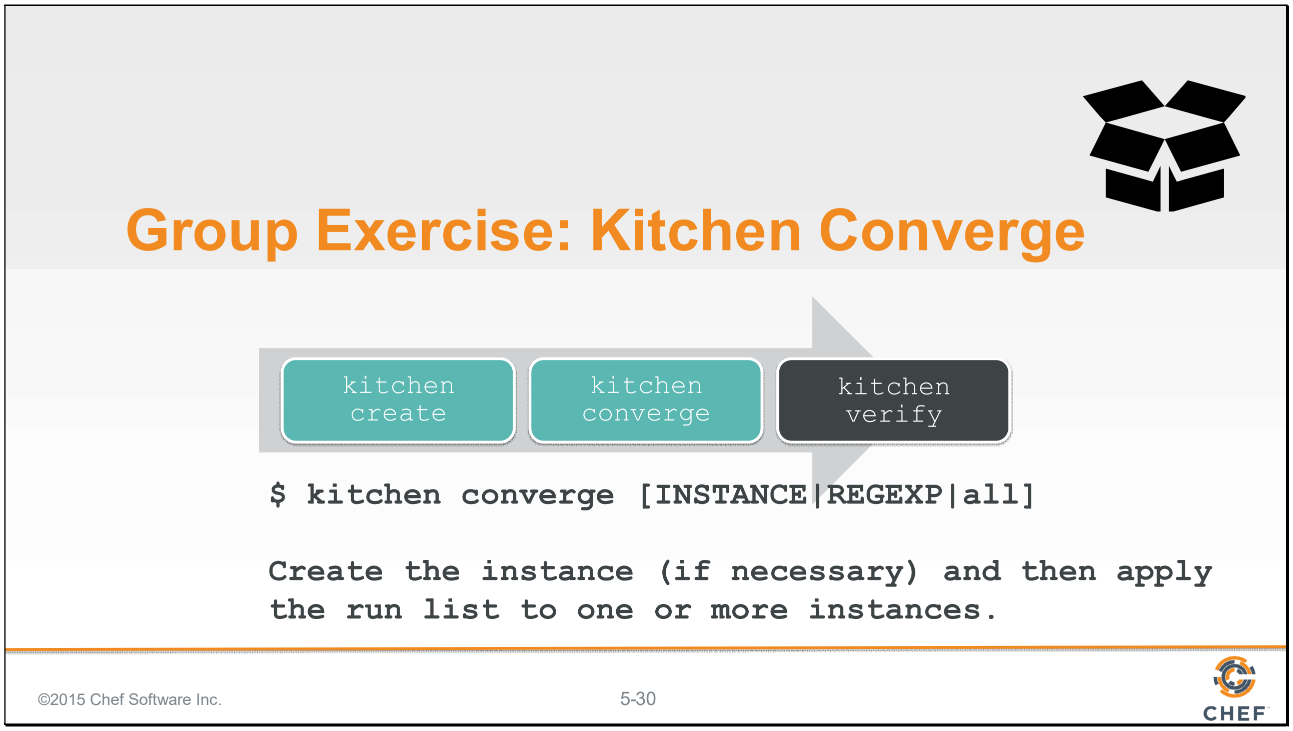
Running `kitchen create default-centos-67` would create the the one instance that uses the test suite on the platform we want.

Typing in that name would be tiring if you had a lot of instances. A shortcut can be used to target the same system `kitchen create default` or `kitchen create centos` or even `kitchen create 67`. This is an example of using the Regular Expression (REGEXP) to specify an instance.

When you want to target all of the instances you can run `kitchen create`. This will create all instances. Seeing as how there is only one instance this will work well.

In our case, this command would use the Docker driver to create a docker image based on centos-6.7.

Slide 30



Group Exercise: Kitchen Converge

kitchen create
kitchen converge
kitchen verify

```
$ kitchen converge [INSTANCE|REGEXP|all]
```

Create the instance (if necessary) and then apply the run list to one or more instances.

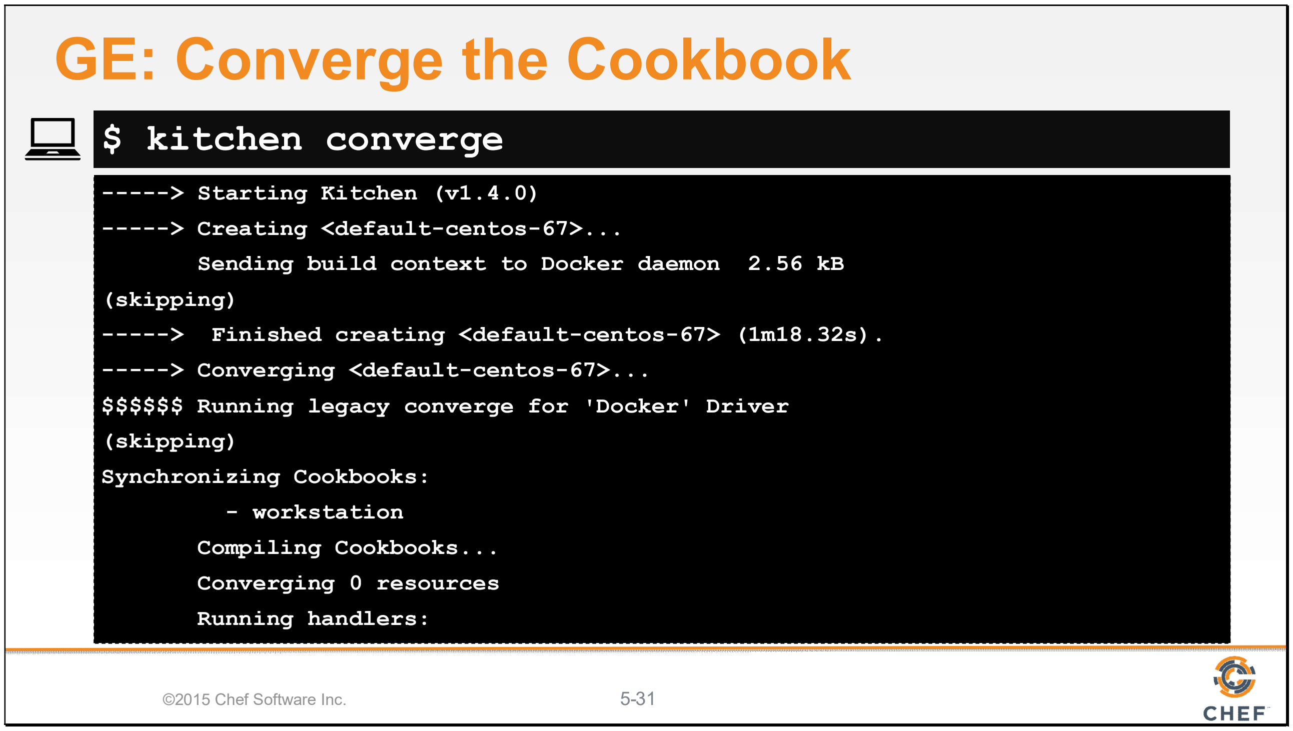
Creating an image gives us a instance to test our cookbooks but it still would leave us with the work of installing chef and applying the cookbook defined in our .kitchen.yml run list.

So let's introduce you to the second kitchen command: `kitchen converge`.

Converging an instance will create the instance if it has not already been created. Then it will install chef and apply that cookbook to that instance.

In our case, this command would take our image and install chef and apply the workstation cookbook's default recipe.

Slide 31



GE: Converge the Cookbook

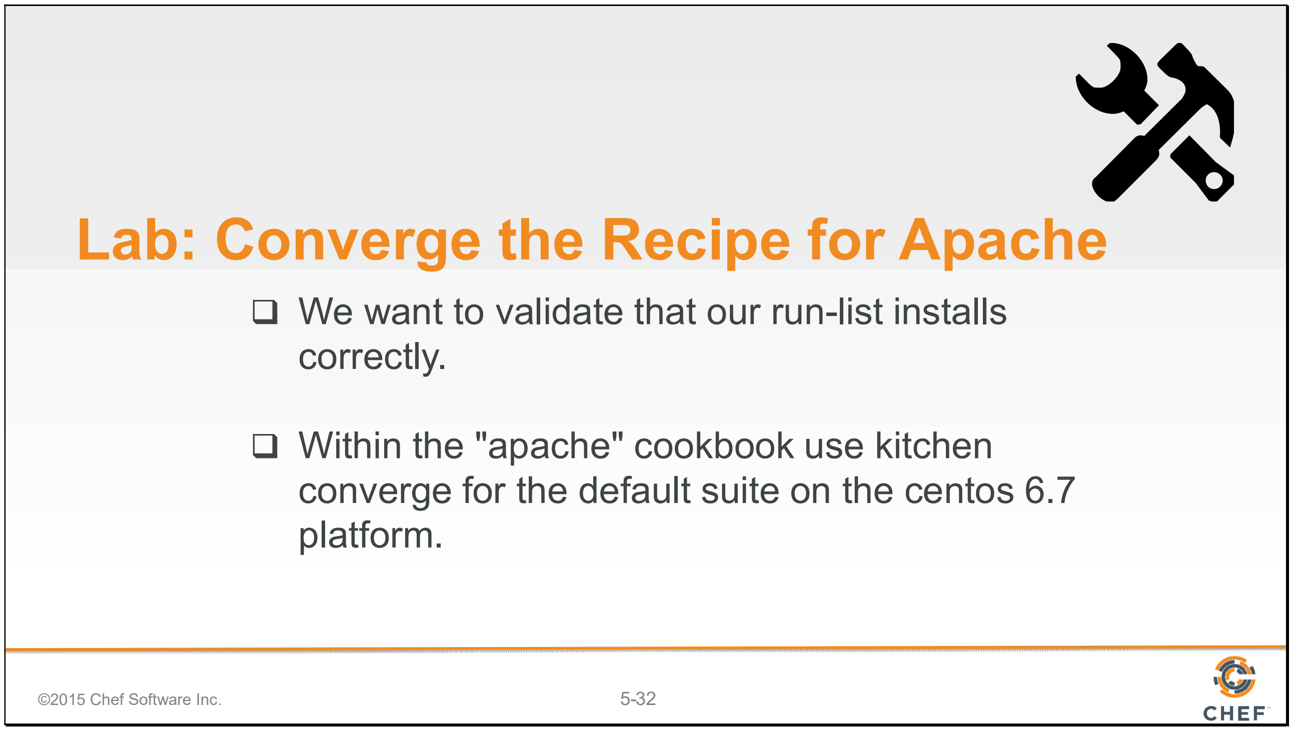
```
$ kitchen converge
```

```
-----> Starting Kitchen (v1.4.0)
-----> Creating <default-centos-67>...
       Sending build context to Docker daemon  2.56 kB
(skipping)
----->  Finished creating <default-centos-67> (1m18.32s).
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
(skipping)
Synchronizing Cookbooks:
         - workstation
       Compiling Cookbooks...
       Converging 0 resources
       Running handlers:
```

Be sure you are at ~/cookbooks/workstation and then run `kitchen converge` to verify that the workstation cookbook is able to converge the default recipe against the platform centos 6.7.

The workstation cookbook should successfully apply the default recipe. If an error occurs, let's stop and troubleshoot the issues.

Slide 32

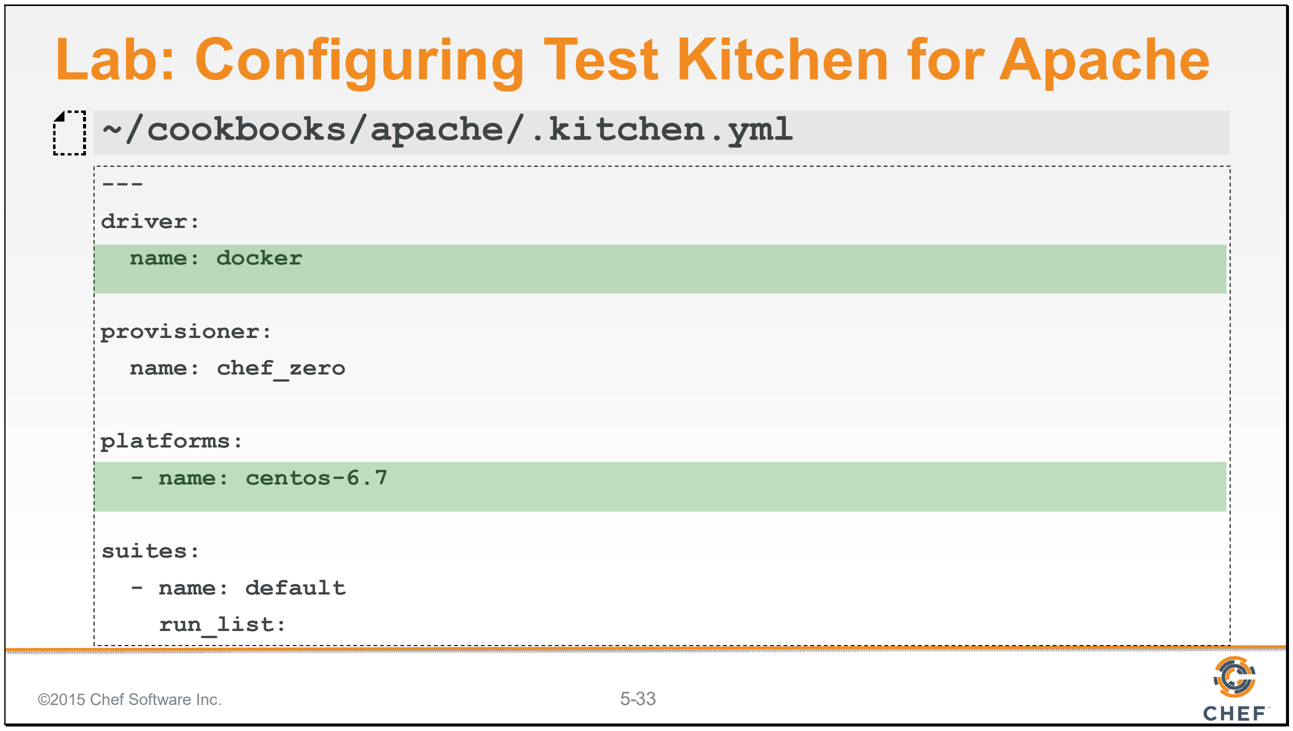


Lab: Converge the Recipe for Apache

- ❑ We want to validate that our run-list installs correctly.
- ❑ Within the "apache" cookbook use kitchen converge for the default suite on the centos 6.7 platform.

Do the same thing again for the apache cookbook. Update the .kitchen.yml file so that it converges the apache cookbook's default recipe on the centos-6.7 platform with the docker driver.

Slide 33



Lab: Configuring Test Kitchen for Apache

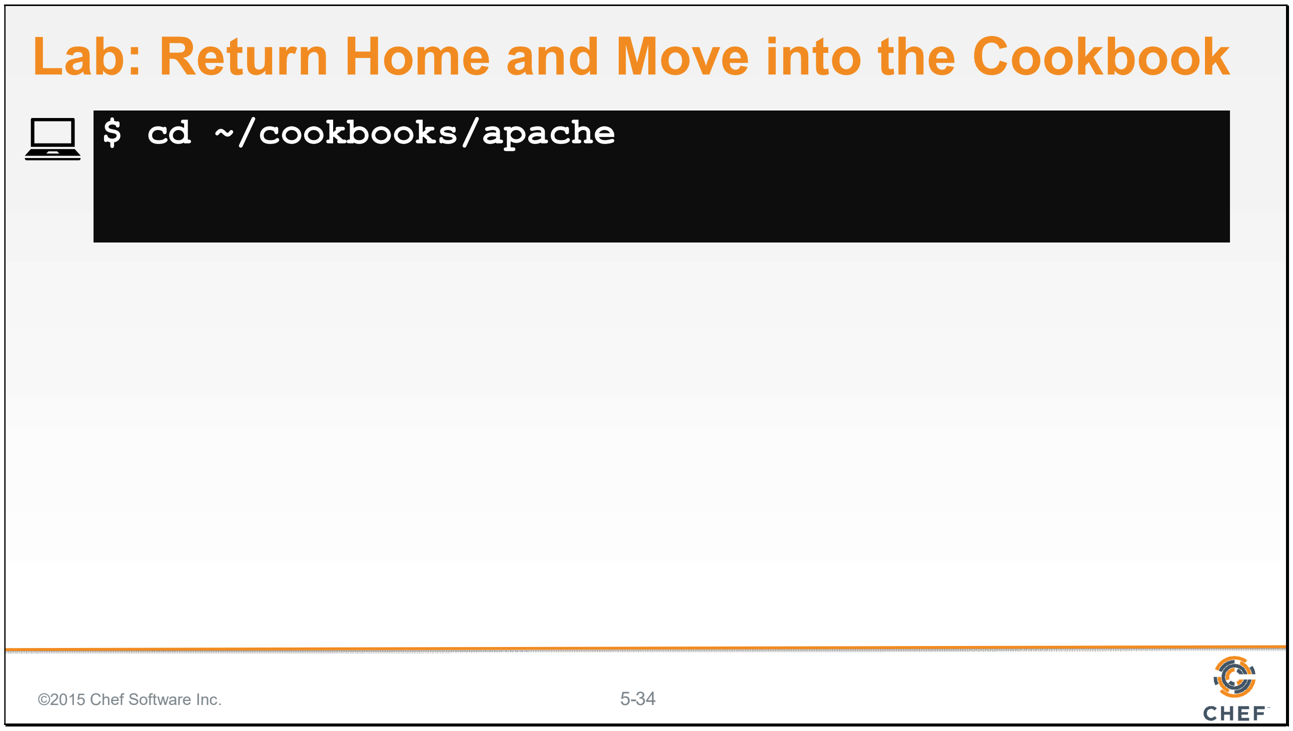
~/cookbooks/apache/.kitchen.yml

```
---
driver:
  name: docker

provisioner:
  name: chef_zero

platforms:
  - name: centos-6.7

suites:
  - name: default
    run_list:
```

©2015 Chef Software Inc. 5-33

Like you did before, update the .kitchen.yml file to use the docker driver and the centos-6.7 platform.

Slide 34

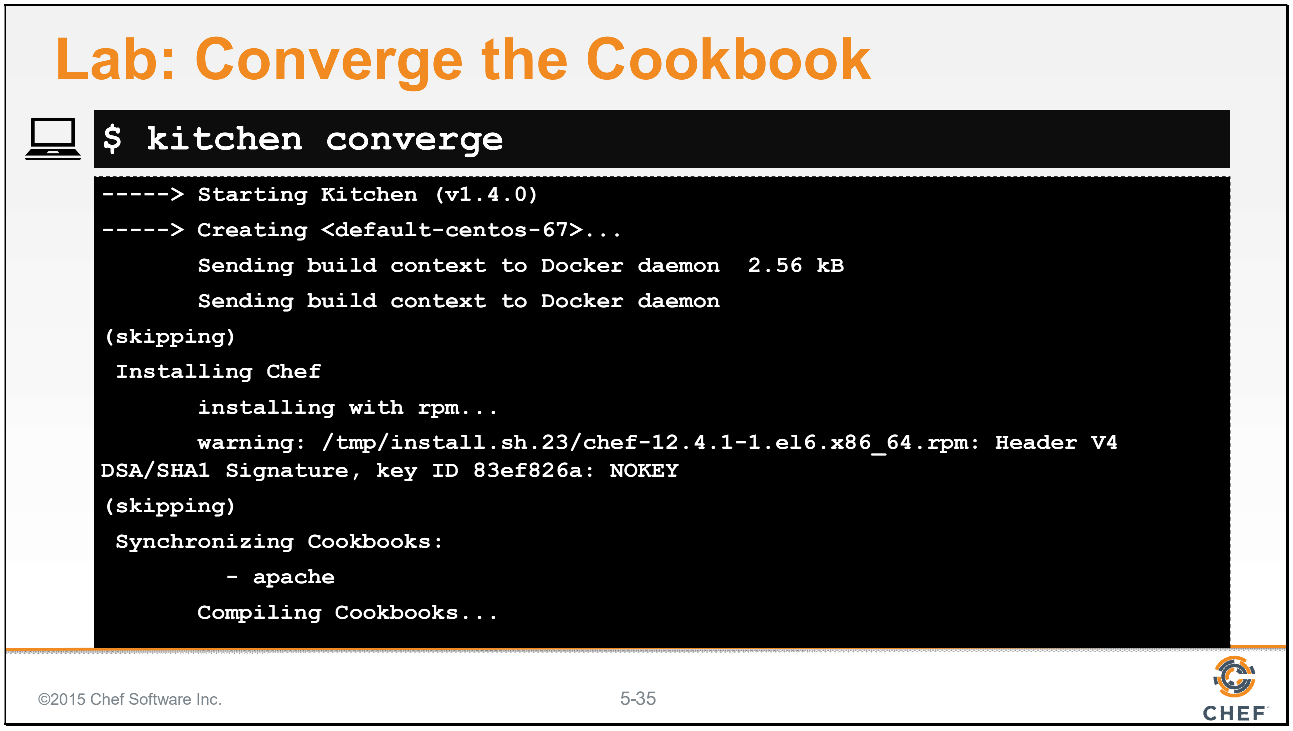


Lab: Return Home and Move into the Cookbook

```
$ cd ~/cookbooks/apache
```

Change into the apache cookbook folder.

Slide 35

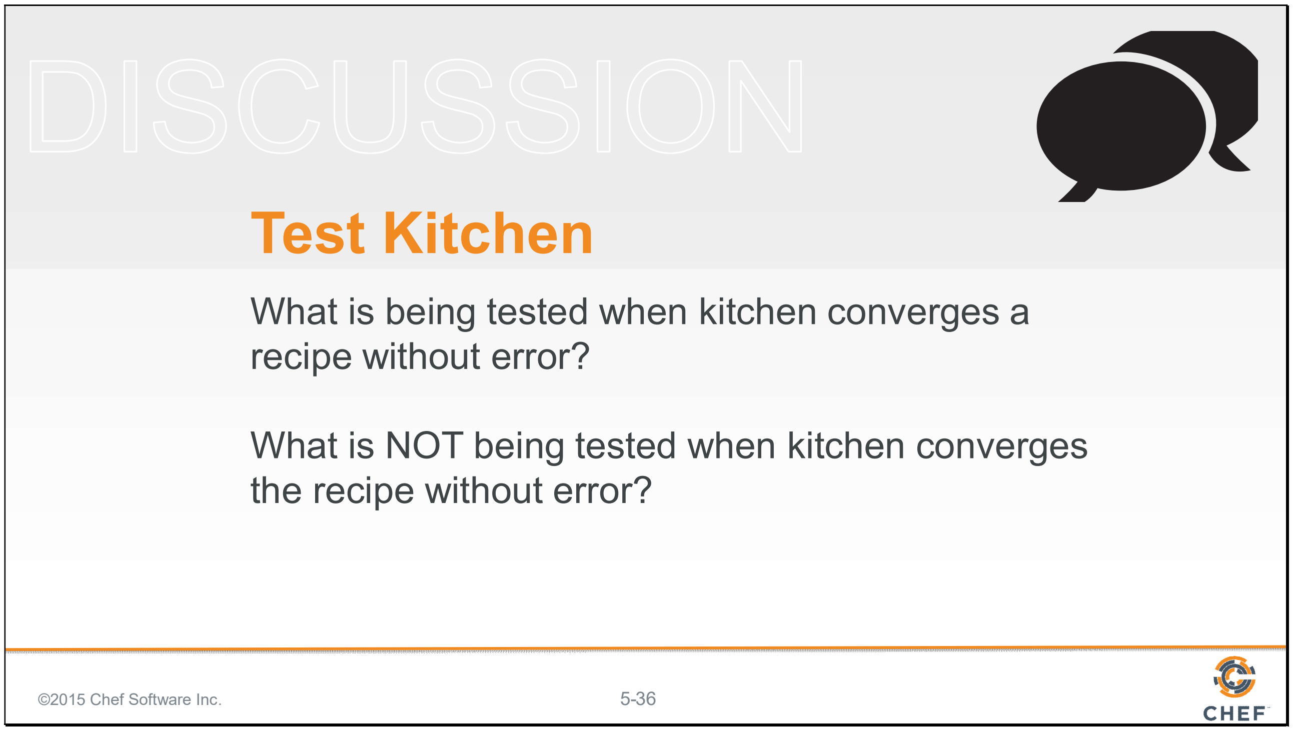


Lab: Converge the Cookbook

```
$ kitchen converge
```

```
-----> Starting Kitchen (v1.4.0)
-----> Creating <default-centos-67>...
       Sending build context to Docker daemon  2.56 kB
       Sending build context to Docker daemon
(skipping)
 Installing Chef
       installing with rpm...
       warning: /tmp/install.sh.23/chef-12.4.1-1.el6.x86_64.rpm: Header V4
DSA/SHA1 Signature, key ID 83ef826a: NOKEY
(skipping)
 Synchronizing Cookbooks:
         - apache
       Compiling Cookbooks...
```

Execute `kitchen converge` to validate that our apache cookbook's default recipe is able to converge on the centos-6.7 instance.



DISCUSSION

Test Kitchen

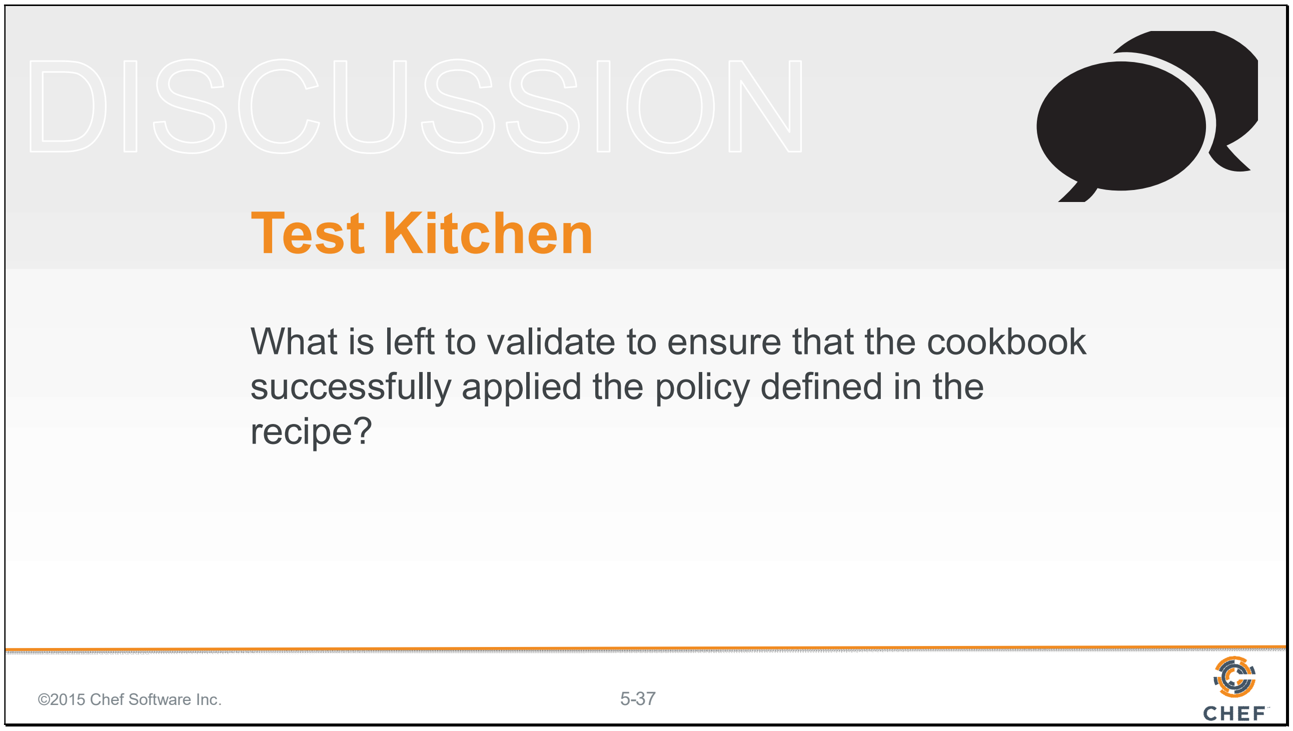
What is being tested when kitchen converges a recipe without error?

What is NOT being tested when kitchen converges the recipe without error?

CHEF

Kitchen converge will create the instance if it is not already created. It will install Chef. Then it applies the recipe to the system examining each of the resources and asking them to take action to place the system into the desired state. What is being tested when kitchen converges a recipe without error?

What is NOT being tested when kitchen converges the recipe without error?



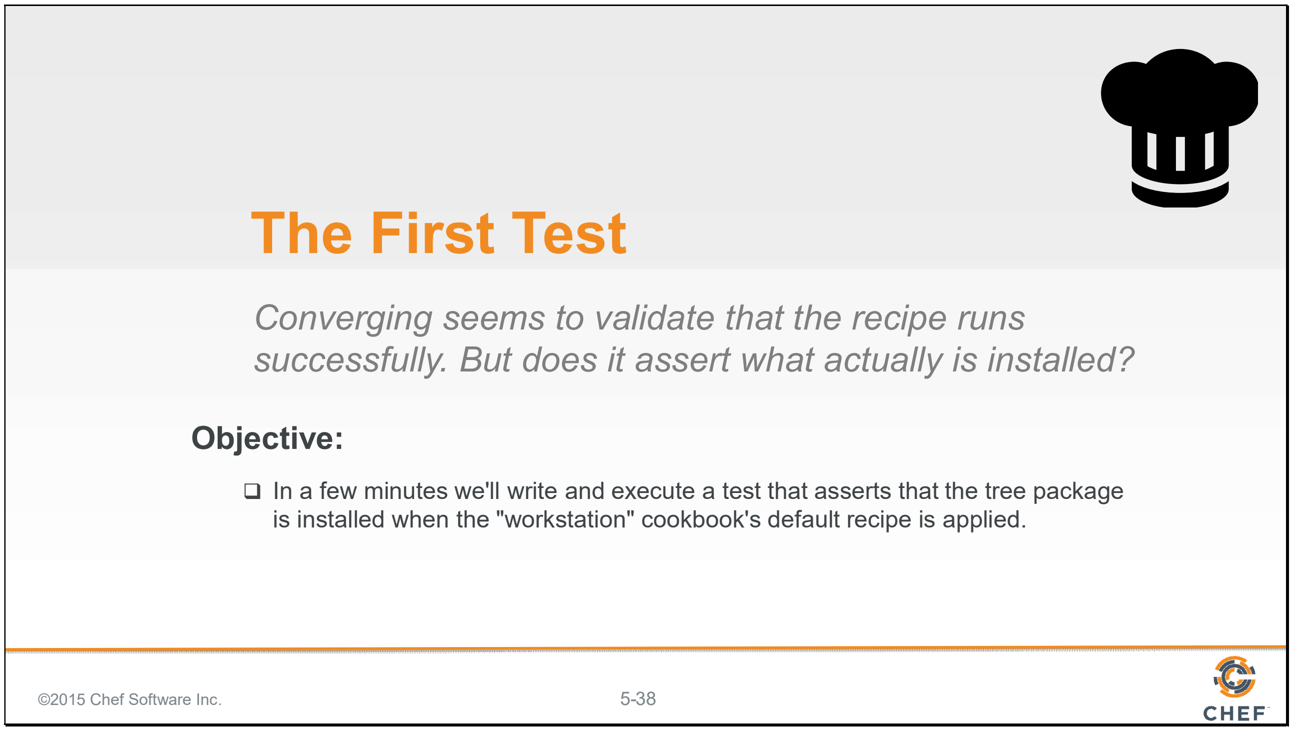
DISCUSSION

Test Kitchen

What is left to validate to ensure that the cookbook successfully applied the policy defined in the recipe?

What is left to validate to ensure that the cookbook successfully applied the policy defined in the recipe?

Slide 38



The First Test

Converging seems to validate that the recipe runs successfully. But does it assert what actually is installed?

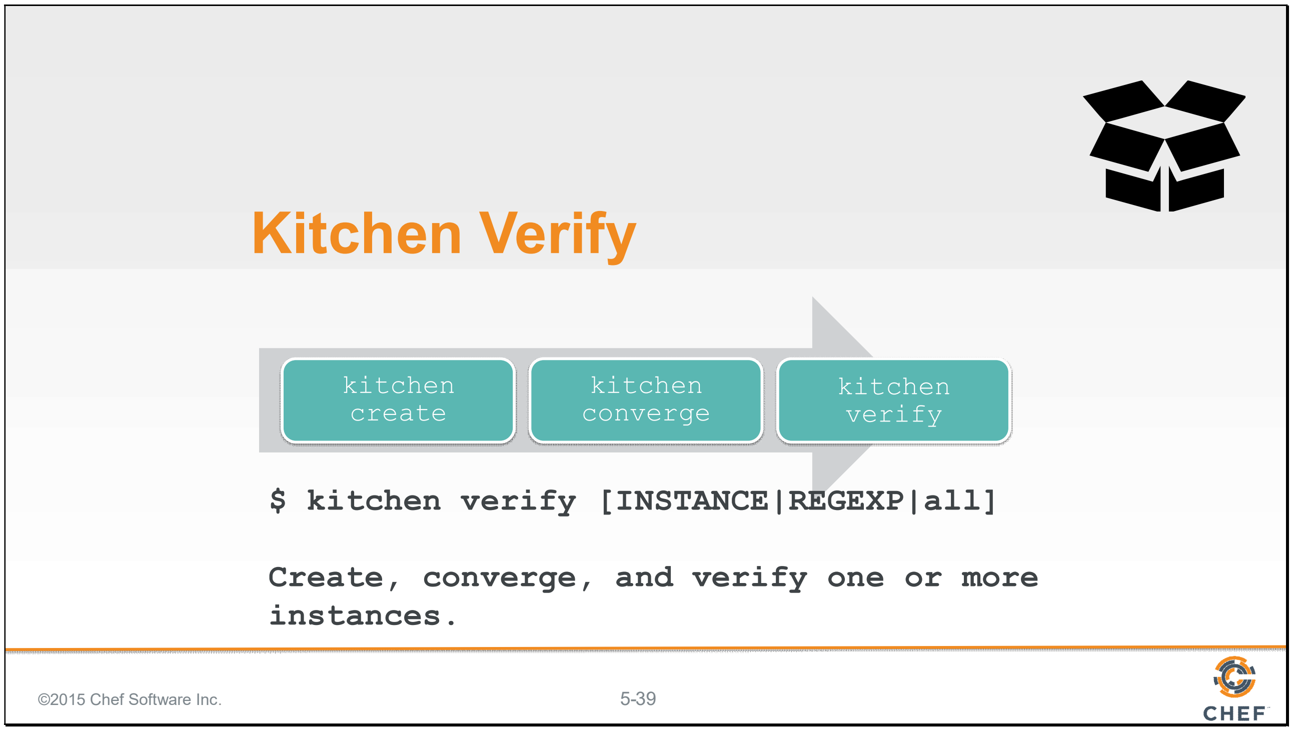
Objective:

- ❑ In a few minutes we'll write and execute a test that asserts that the tree package is installed when the "workstation" cookbook's default recipe is applied.

There is no automation that automatically understands the intention defined in the recipes we create. To do that we will define our own automated test.

Let’s explore testing by adding a simple test to validate that the tree package is installed after converging the workstation cookbook's default recipe. We'll do this together in a few minutes.

Slide 39



Kitchen Verify

kitchen create
kitchen converge
kitchen verify

```
$ kitchen verify [INSTANCE|REGEXP|all]
```

```
Create, converge, and verify one or more
instances.
```

The third kitchen command is `kitchen verify`.

To verify an instance means to:

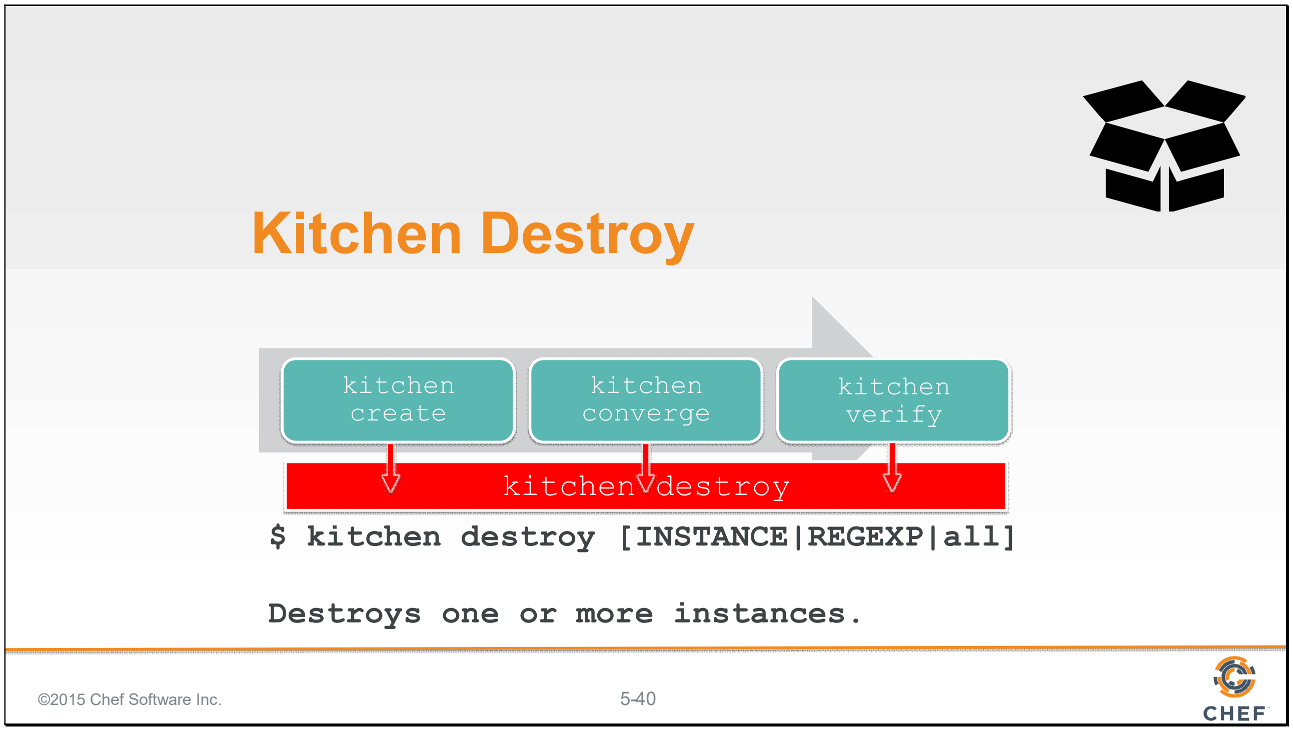
Create a virtual or cloud instances, if needed

Converge the instance, if needed

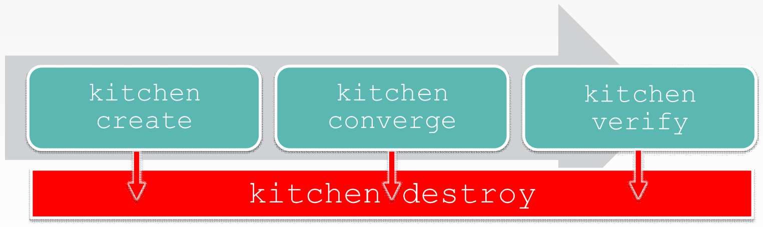
And then execute a collection of defined tests against the instance

In our case, our instance has already been created and converged so when we run `kitchen verify` it will execute the tests that we will later define.

Slide 40



Kitchen Destroy



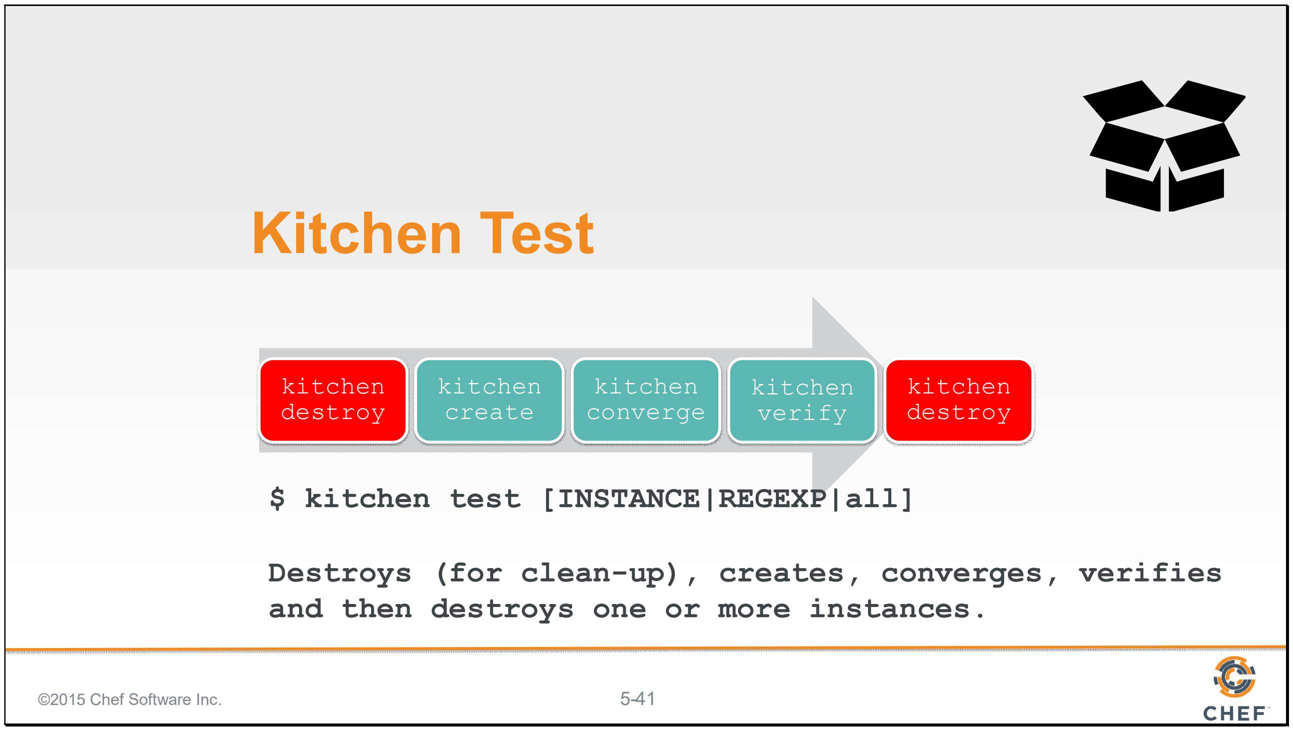
```
$ kitchen destroy [INSTANCE|REGEXP|all]
```

Destroys one or more instances.

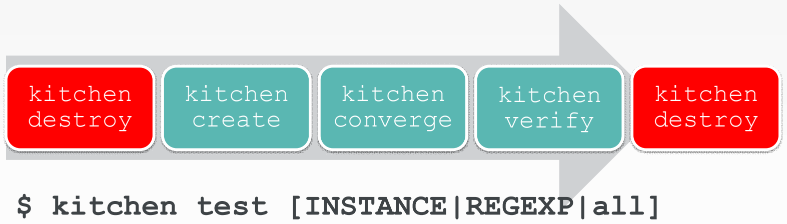
The fourth kitchen command is `kitchen destroy`.

Destroy is available at all stages and essentially cleans up the instance.

Slide 41



Kitchen Test



```
$ kitchen test [INSTANCE|REGEXP|all]
```

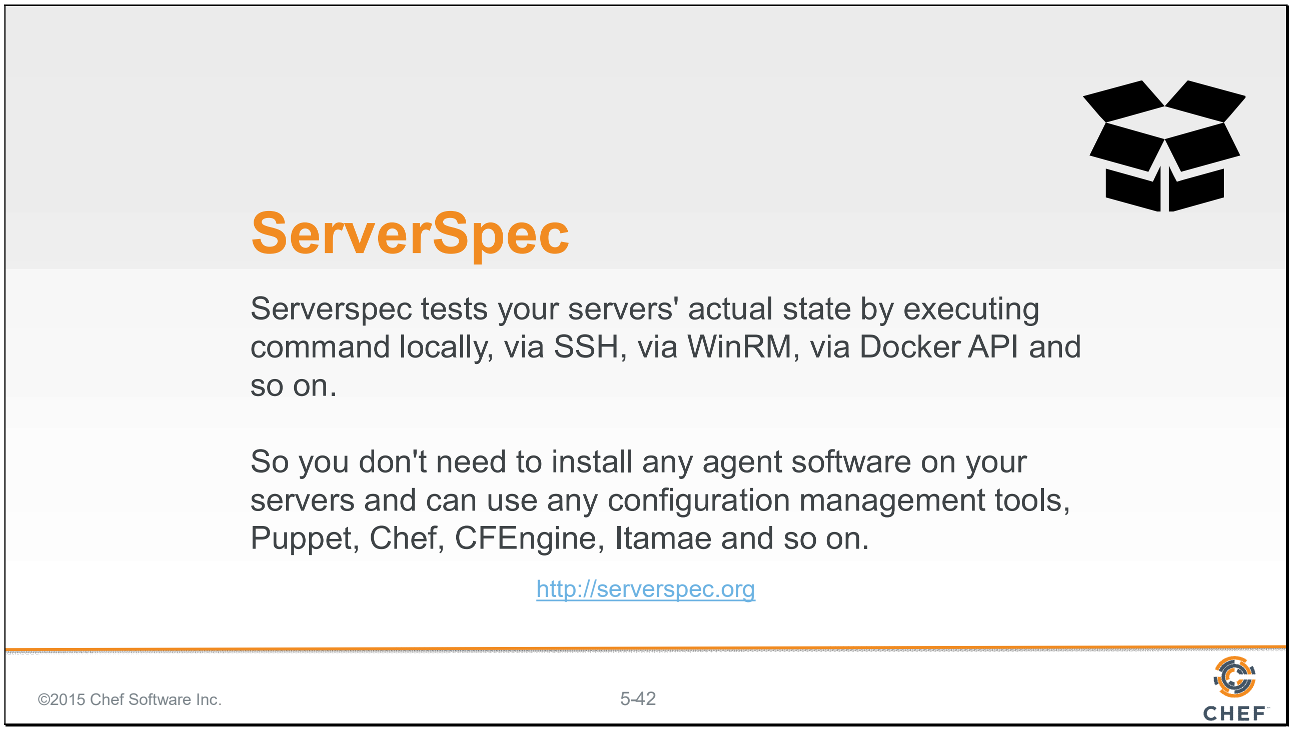
Destroys (for clean-up), creates, converges, verifies and then destroys one or more instances.

There a single command that encapsulates the entire workflow - that is `kitchen test`.

Kitchen test ensures that if the instance was in any state - created, converged, or verified - that it is immediately destroyed. This ensures a clean instance to perform all of the steps: create; converge; and verify. `kitchen test` completes the entire execution by destroying the instance at the end.

Traditionally this all encompassing workflow is useful to ensure that we have a clean state when we start and we do not leave a mess behind us.

Slide 42



ServerSpec

Serverspec tests your servers' actual state by executing command locally, via SSH, via WinRM, via Docker API and so on.

So you don't need to install any agent software on your servers and can use any configuration management tools, Puppet, Chef, CFEngine, Itamae and so on.

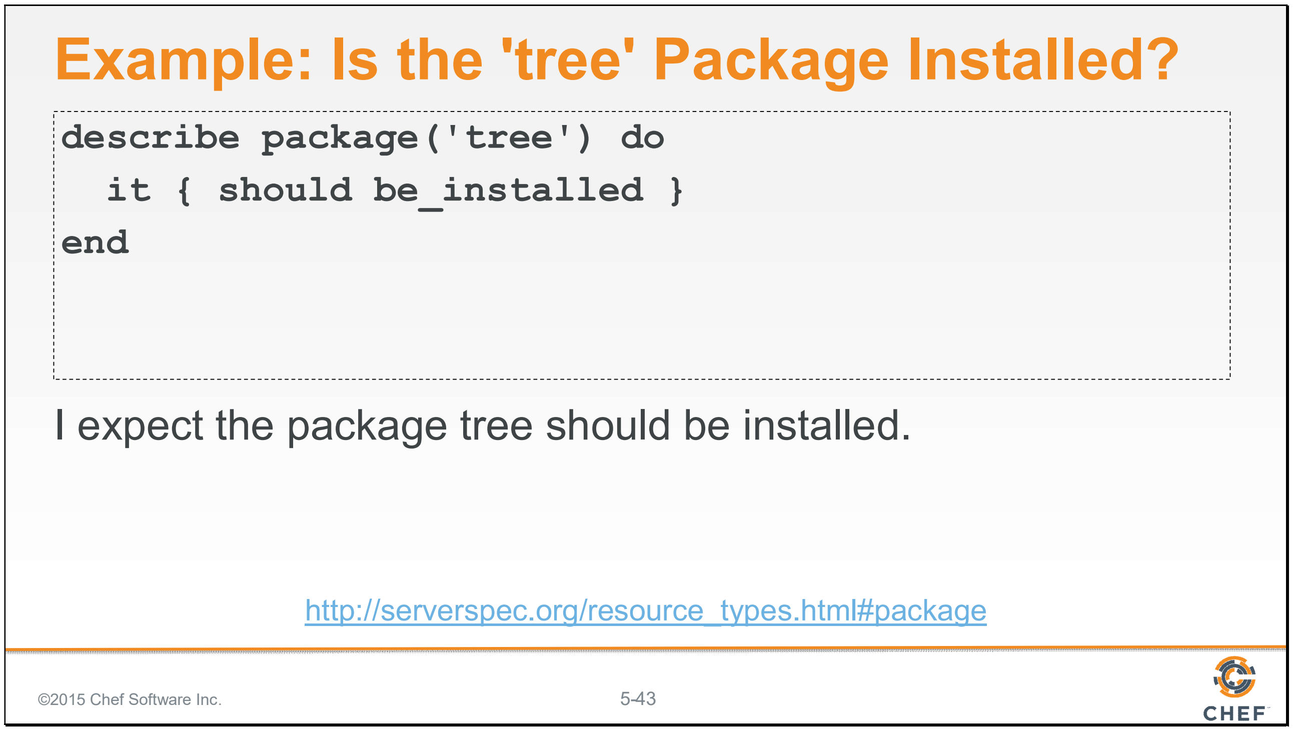
http://serverspec.org

So `kitchen verify` and `kitchen test` are the two kitchen commands that we can use to execute a body of tests against our instances. Now it is time to define those tests with ServerSpec.

ServerSpec is one of many possible test frameworks that Test Kitchen supports. It is a popular choice for those doing Chef cookbook development because ServerSpec is built on a Ruby testing framework named RSpec.

RSpec is similar to Chef - as it is a Domain Specific Language, or DSL, layered on top of Ruby. Where Chef gives us a DSL to describe the policy of our system, RSpec allows us to describe the expectations of tests that we define. ServerSpec adds a number of helpers to RSpec to make it easy to test the state of a system.

Slide 43



Example: Is the 'tree' Package Installed?

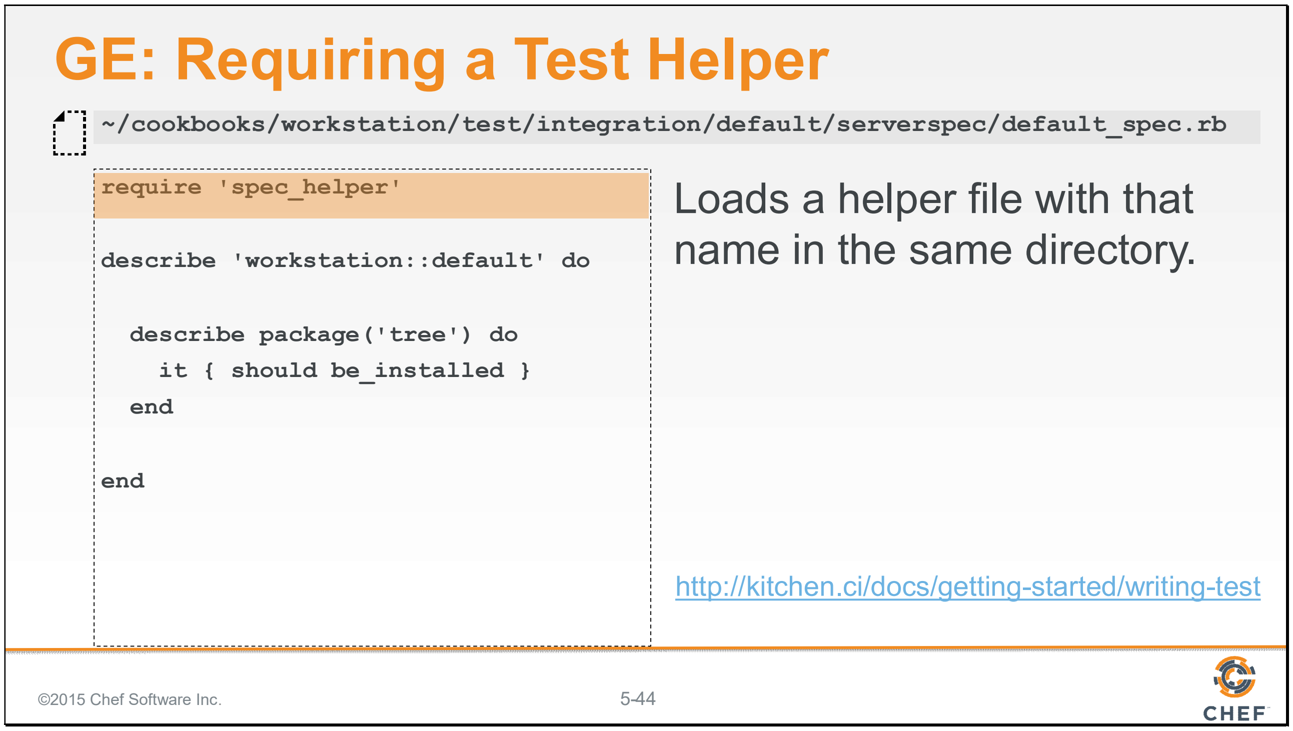
```
describe package('tree') do
  it { should be_installed }
end
```

I expect the package tree should be installed.

http://serverspec.org/resource_types.html#package

Here is an example of an isolated ServerSpec expectation that states: We expect the package named 'tree' to be installed.

Slide 44



GE: Requiring a Test Helper

~/cookbooks/workstation/test/integration/default/serverspec/default_spec.rb

```
require 'spec_helper'

describe 'workstation::default' do

  describe package('tree') do
    it { should be_installed }
  end

end
```

Loads a helper file with that name in the same directory.

http://kitchen.ci/docs/getting-started/writing-test

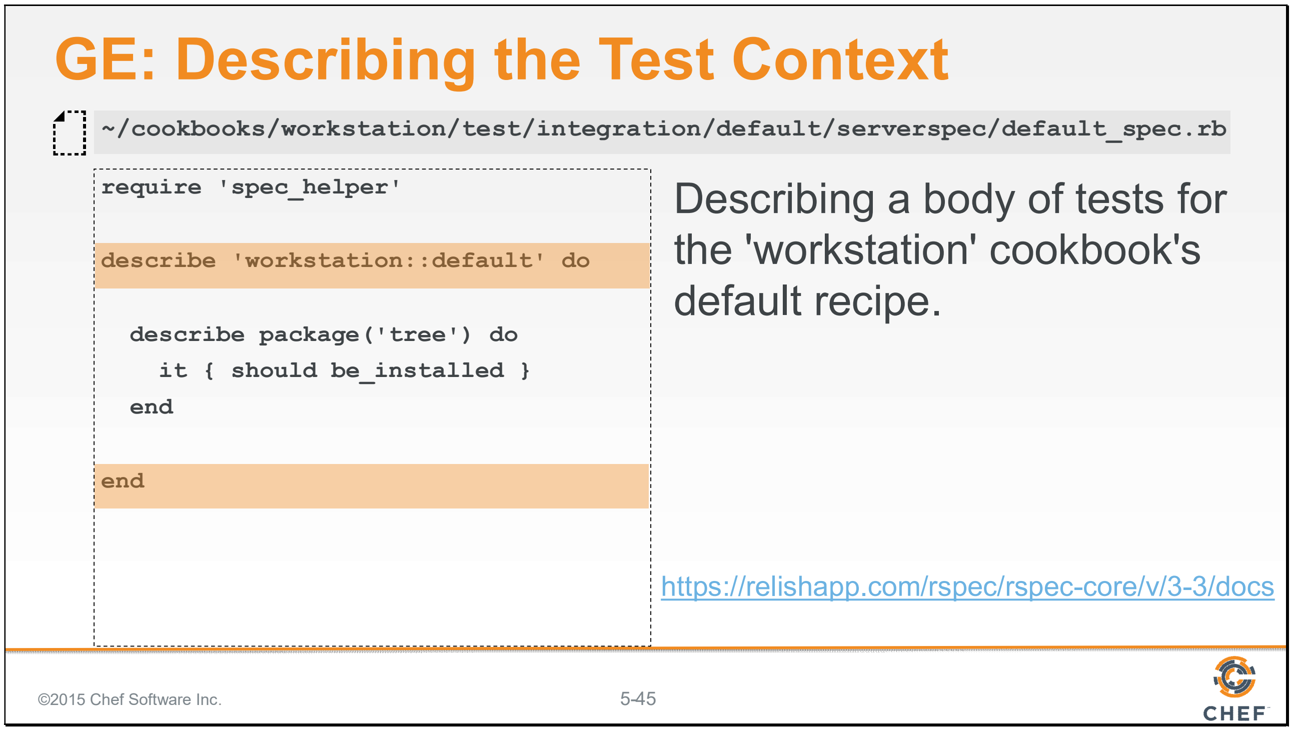
©2015 Chef Software Inc. 5-44

For our test to work with Test Kitchen there are a number of conventions that we need to adhere to have our test code load correctly.

First, we need to create a test file, often referred to as a spec file at the following path. The structure of the path is a convention defined by Test Kitchen and will automatically be loaded when we run `kitchen verify`. Fortunately for us the test file has already been created when we used 'chef' to generate the workstation cookbook.

Within the spec file we need to first require a helper file. The helper is were we keep common helper methods and library requires in one location. This is likely already present within the generated test file.

Slide 45



GE: Describing the Test Context

~/cookbooks/workstation/test/integration/default/serverspec/default_spec.rb

```
require 'spec_helper'

describe 'workstation::default' do

  describe package('tree') do
    it { should be_installed }
  end

end
```

Describing a body of tests for the 'workstation' cookbook's default recipe.

https://relishapp.com/rspec/rspec-core/v/3-3/docs

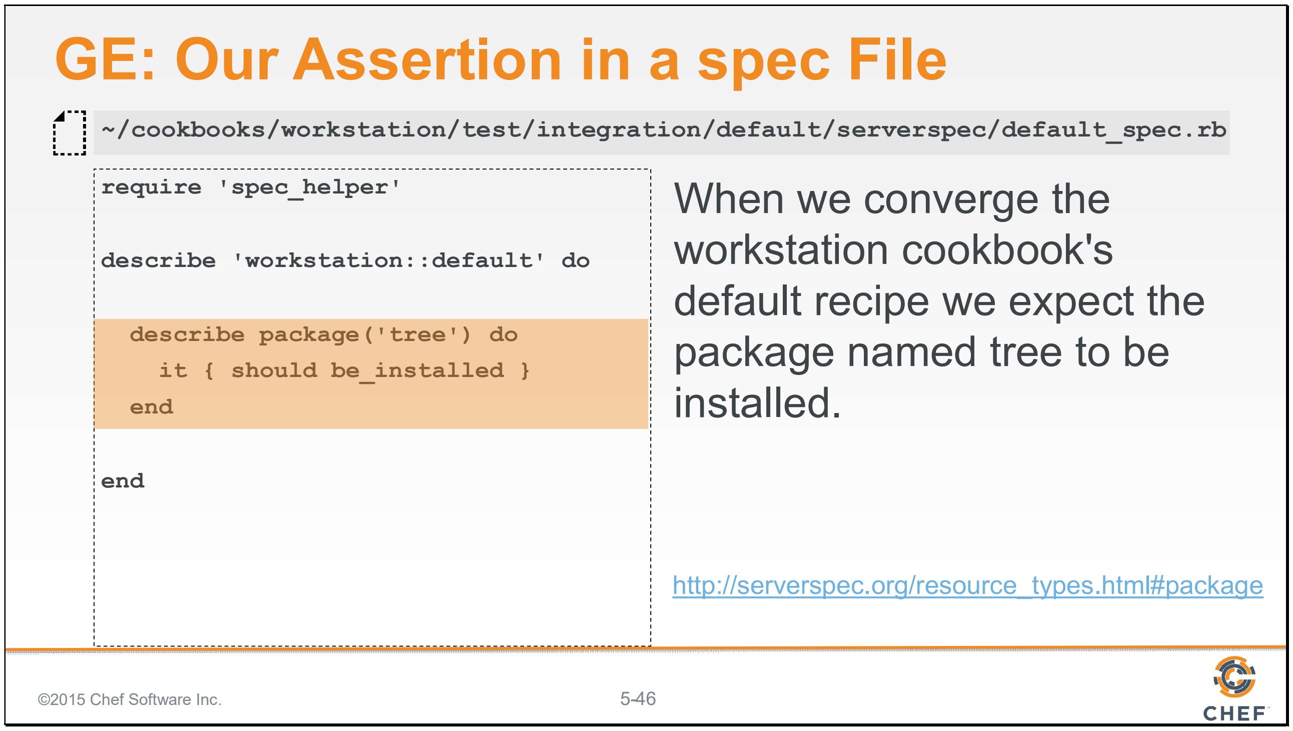
©2015 Chef Software Inc. 5-45

Second, we define a describe method. RSpec, which ServerSpec is built on uses an english-like syntax to help us describe the various scenarios and examples that we are testing.

The 'describe' method takes two parameters - the first is the name of fully-qualifed recipe to execute (cookbook_name::recipe_name).

The second parameter is the block between the **do** and **end**. Within that block we can define more describe blocks that allow us to further refine the scenario we are testing.

Slide 46



GE: Our Assertion in a spec File

```
~/cookbooks/workstation/test/integration/default/serverspec/default_spec.rb
```

```ruby
require 'spec_helper'

describe 'workstation::default' do

  describe package('tree') do
    it { should be_installed }
  end

end
```

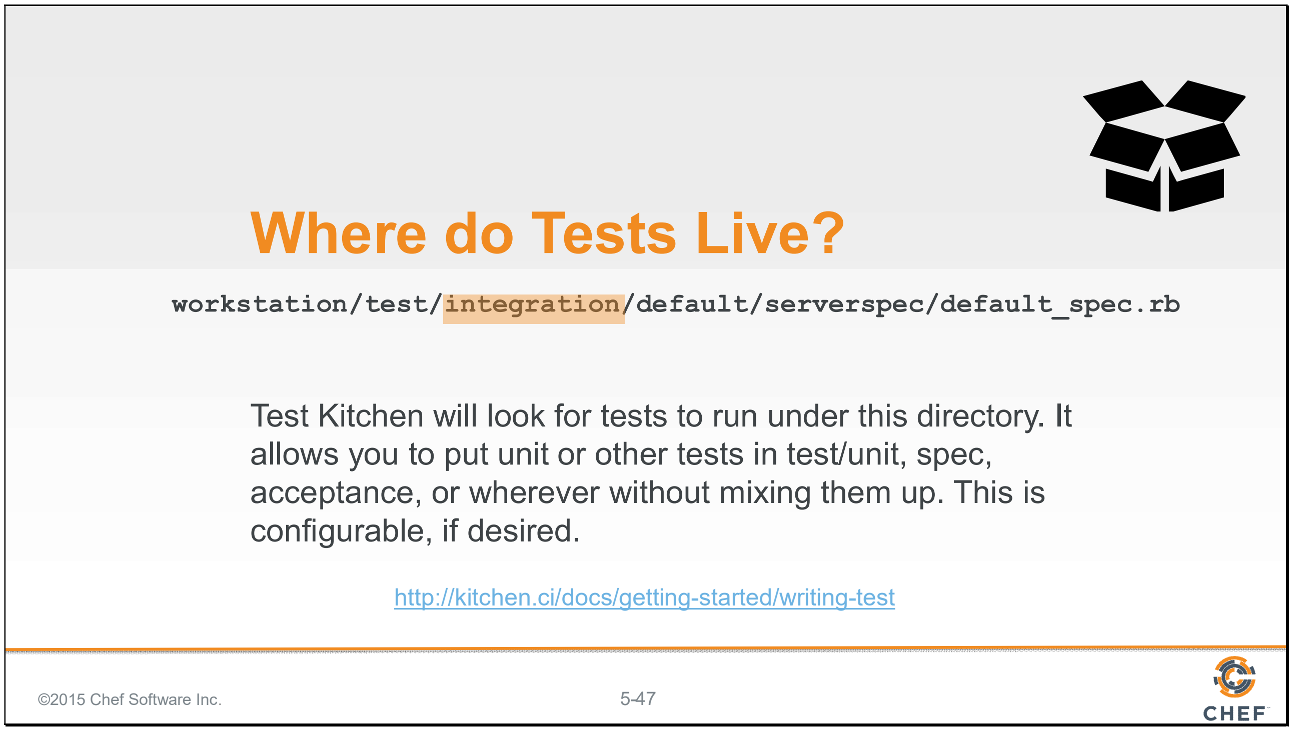
When we converge the workstation cookbook's default recipe we expect the package named tree to be installed.

http://serverspec.org/resource_types.html#package

Here is that example expectation that we showed you earlier except now it is displayed here within this context. This states that when we converge the workstation cookbook's default recipe we want to assert that the tree package has been installed.

Add this expectation to the specification file at the specified path.

Slide 47



Where do Tests Live?

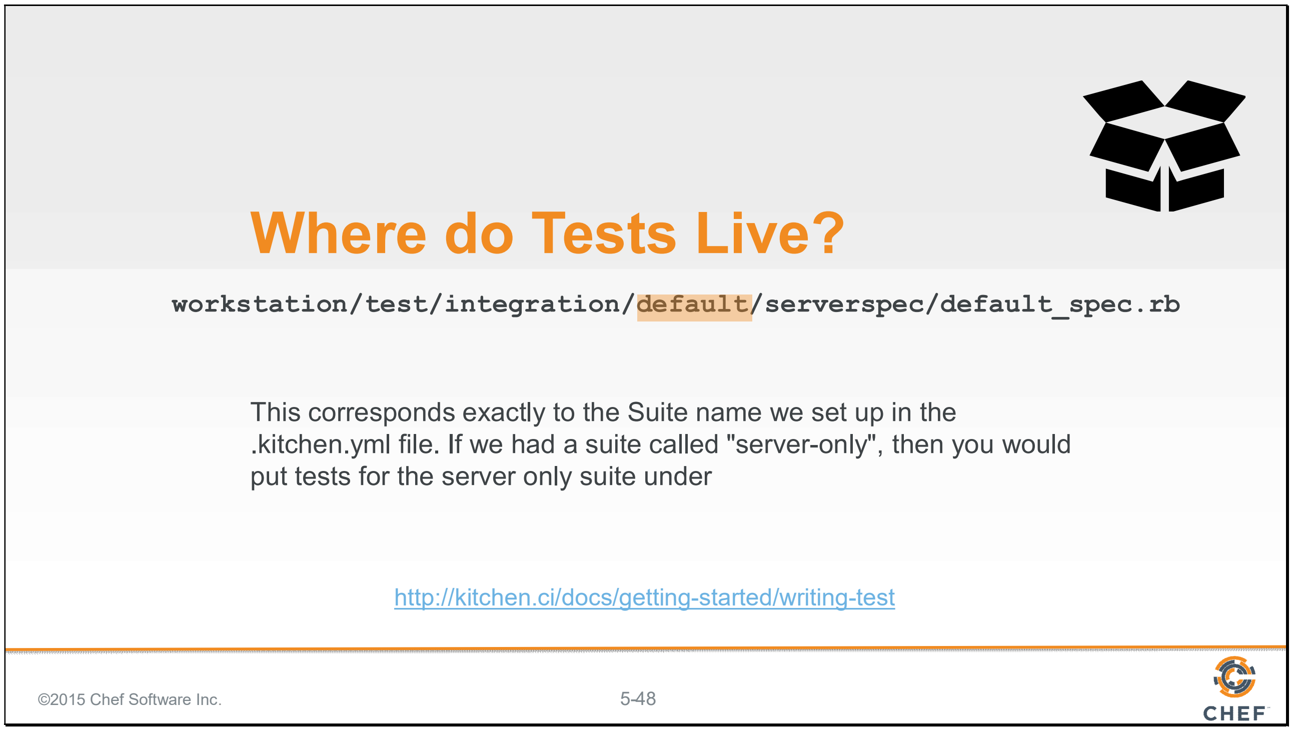
```
workstation/test/integration/default/serverspec/default_spec.rb
```

Test Kitchen will look for tests to run under this directory. It allows you to put unit or other tests in test/unit, spec, acceptance, or wherever without mixing them up. This is configurable, if desired.

http://kitchen.ci/docs/getting-started/writing-test

Let's take a moment to describe the reason behind this long directory path. Within our cookbook we define a test directory and within that test directory we define another directory named 'integration'. This is the basic file path that Test Kitchen expects to find the specifications that we have defined.

Slide 48



Where do Tests Live?

```
workstation/test/integration/default/serverspec/default_spec.rb
```

This corresponds exactly to the Suite name we set up in the .kitchen.yml file. If we had a suite called "server-only", then you would put tests for the server only suite under

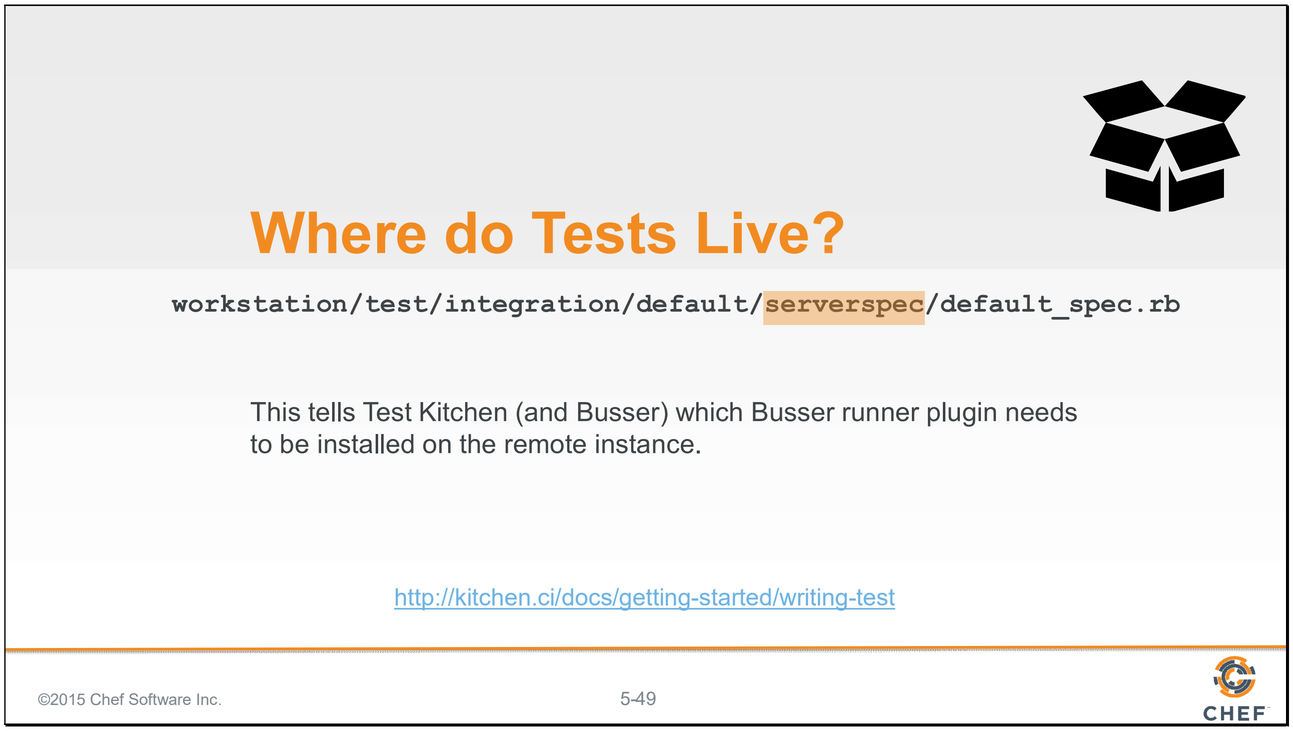
http://kitchen.ci/docs/getting-started/writing-test

©2015 Chef Software Inc. 5-48

CHEF

The next part the path, 'default', corresponds to the name of the test suite that is defined in the .kitchen.yml file. In our case the name of the suite is 'default' so when test kitchen performs a `kitchen verify` for the default suite it will look within the 'default' folder for the specifications to run.

Slide 49



Where do Tests Live?

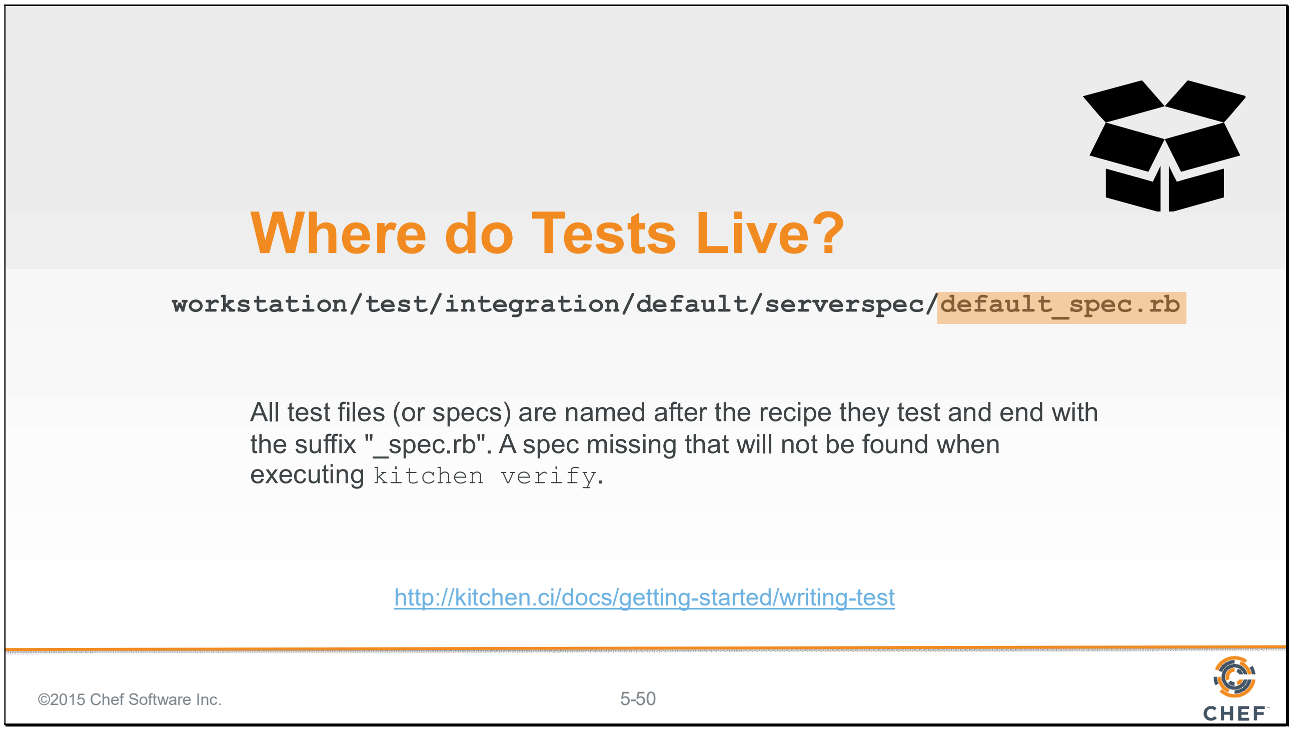
```
workstation/test/integration/default/serverspec/default_spec.rb
```

This tells Test Kitchen (and Busser) which Busser runner plugin needs to be installed on the remote instance.

http://kitchen.ci/docs/getting-started/writing-test

'serverspec' is the kind of tests that we want to define. Test Kitchen supports a number of testing frameworks.

Slide 50



Where do Tests Live?

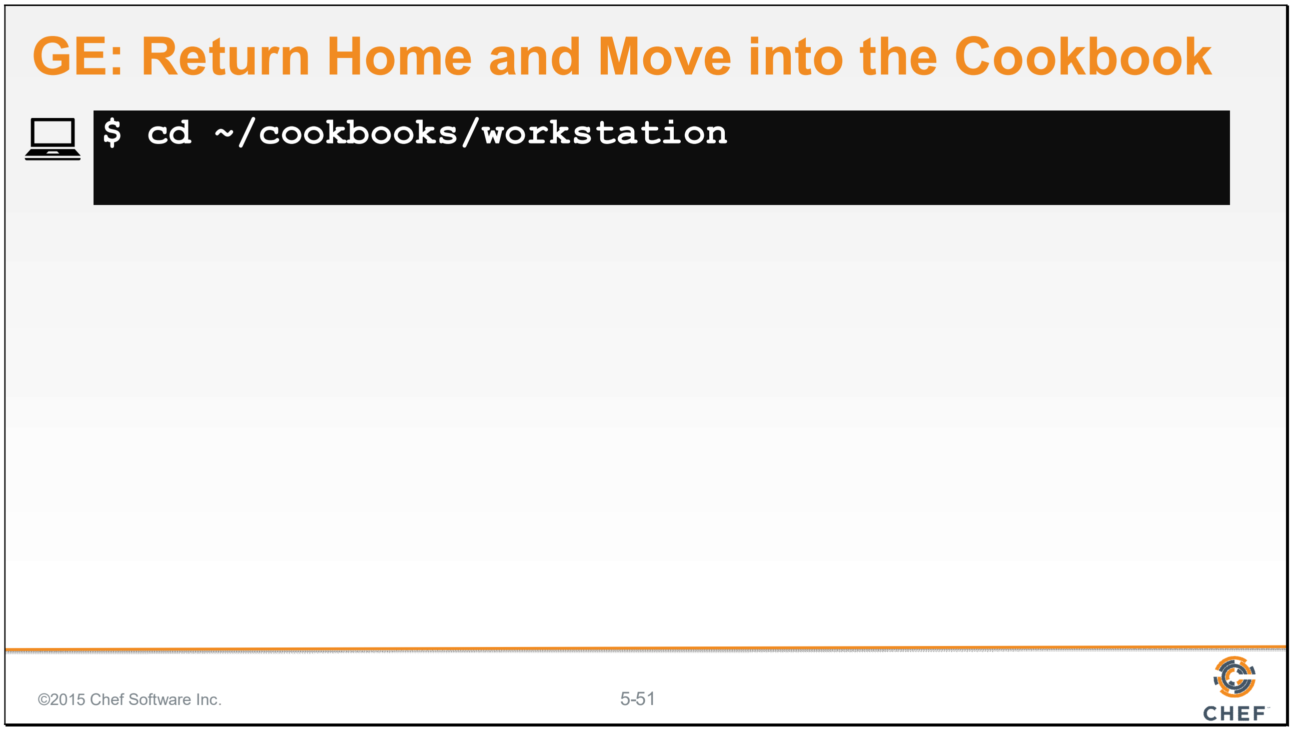
```
workstation/test/integration/default/serverspec/default_spec.rb
```

All test files (or specs) are named after the recipe they test and end with the suffix "_spec.rb". A spec missing that will not be found when executing `kitchen verify`.

http://kitchen.ci/docs/getting-started/writing-test

The final part of the path is the specification file. This is a ruby file. The naming convention for this file is the recipe name with the appended suffix of _spec.rb. All specification files must end with _spec.rb.

Slide 51

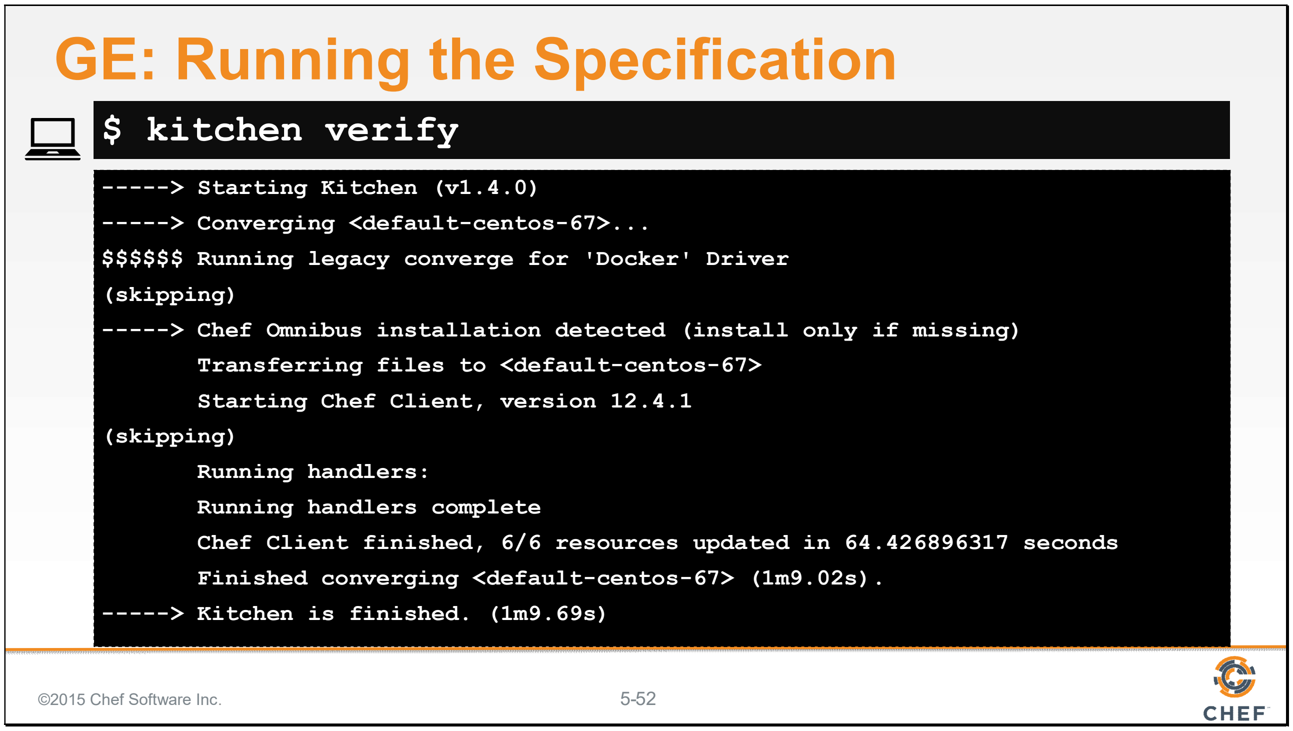


GE: Return Home and Move into the Cookbook

```
$ cd ~/cookbooks/workstation
```

Change into the workstation cookbook directory.

Slide 52



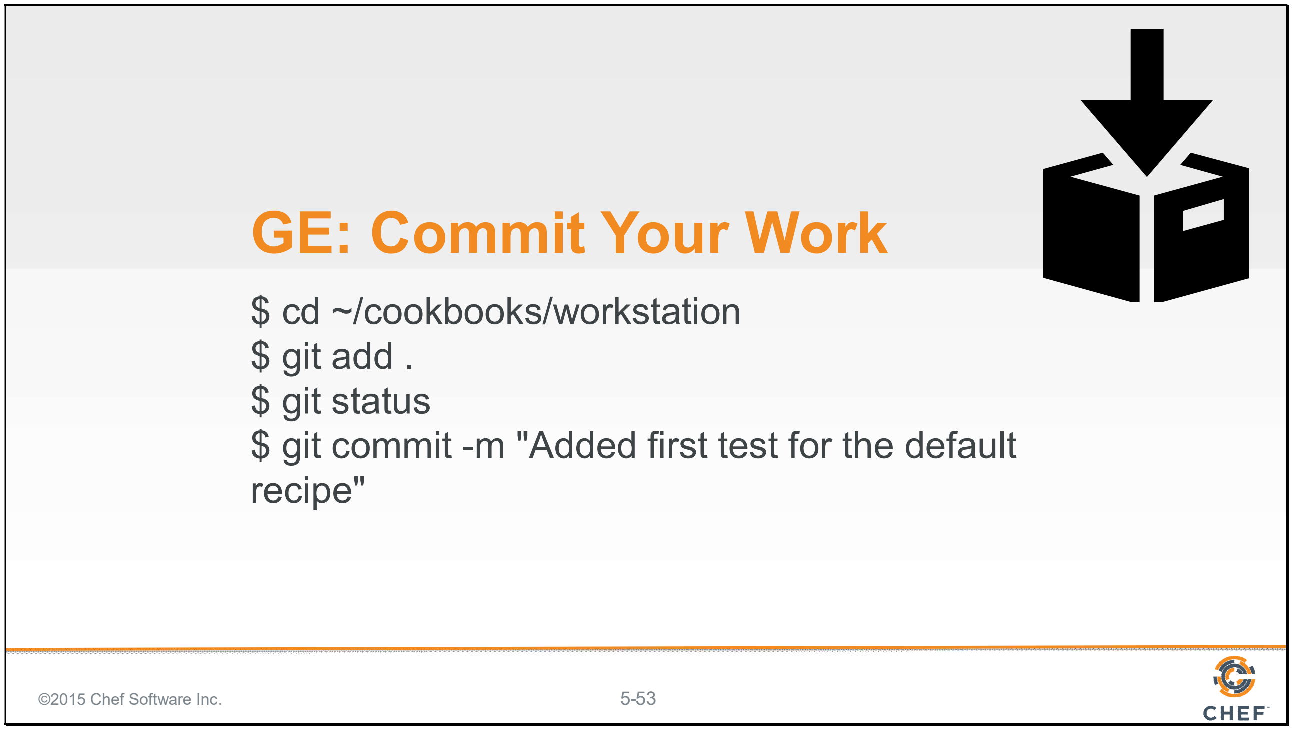
GE: Running the Specification

```
$ kitchen verify
```

```
-----> Starting Kitchen (v1.4.0)
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
(skipping)
-----> Chef Omnibus installation detected (install only if missing)
       Transferring files to <default-centos-67>
       Starting Chef Client, version 12.4.1
(skipping)
       Running handlers:
       Running handlers complete
       Chef Client finished, 6/6 resources updated in 64.426896317 seconds
       Finished converging <default-centos-67> (1m9.02s).
-----> Kitchen is finished. (1m9.69s)
```

With the first test created, lets verify that the package named 'tree' is installed when we apply the workstation cookbooks default recipe using the `kitchen verify` command to execute our test

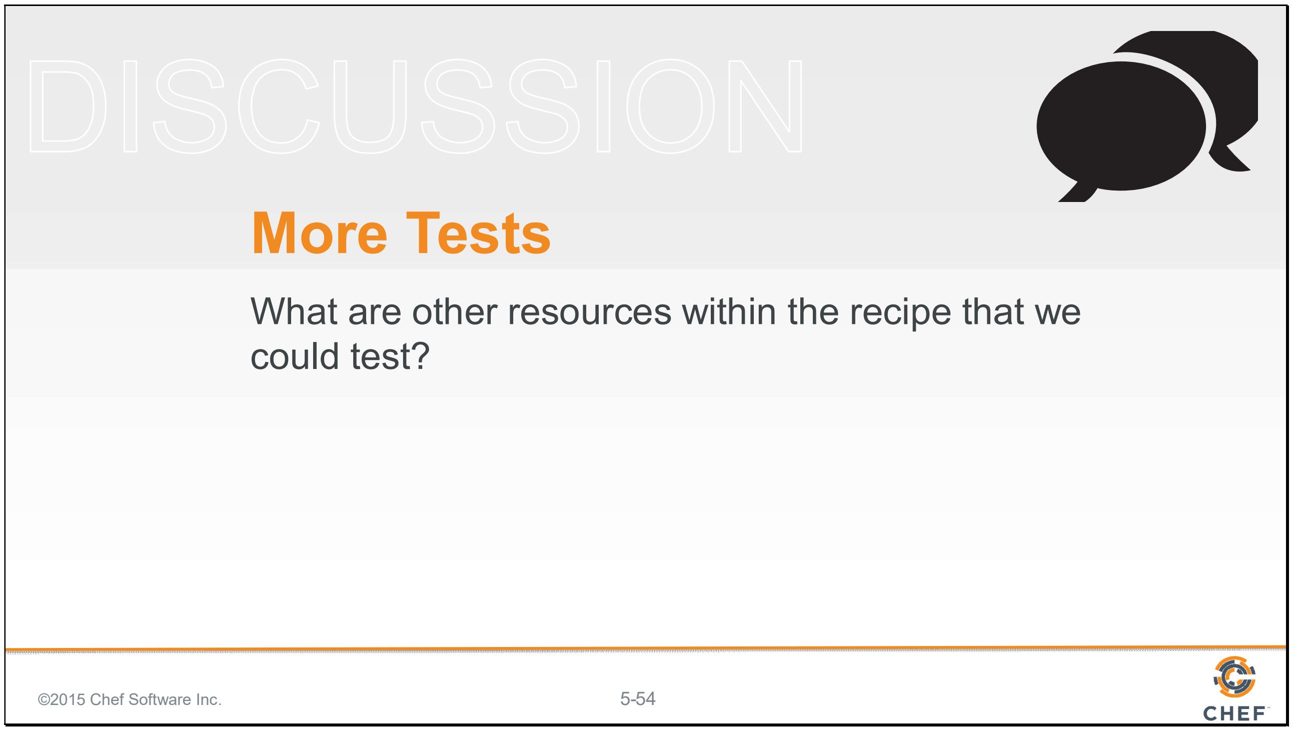
Slide 53



GE: Commit Your Work

```
$ cd ~/cookbooks/workstation
$ git add .
$ git status
$ git commit -m "Added first test for the default recipe"
```

With the first test completed. It is time to commit the changes to source control.



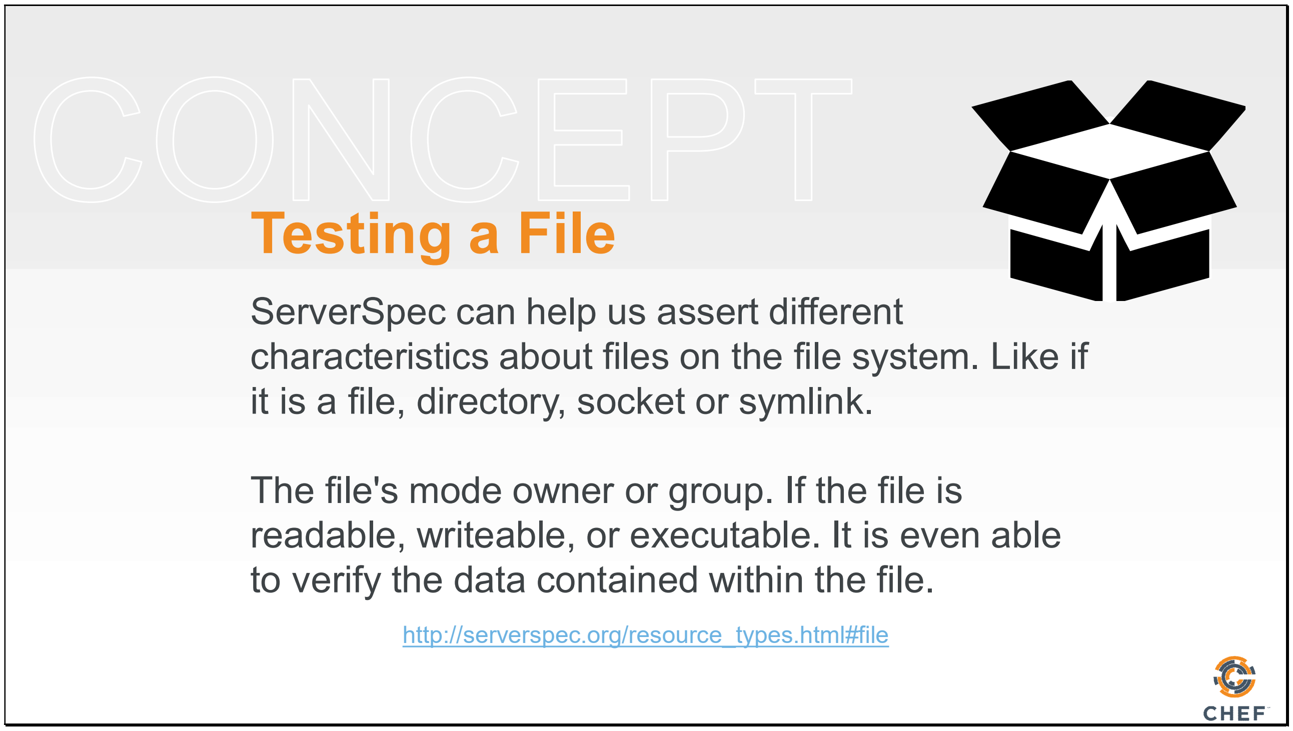
DISCUSSION

More Tests

What are other resources within the recipe that we could test?

Now that we've explored the basic structure of writing tests to validate our cookbook.

What are other resources within the recipe that we could tests?



CONCEPT

Testing a File

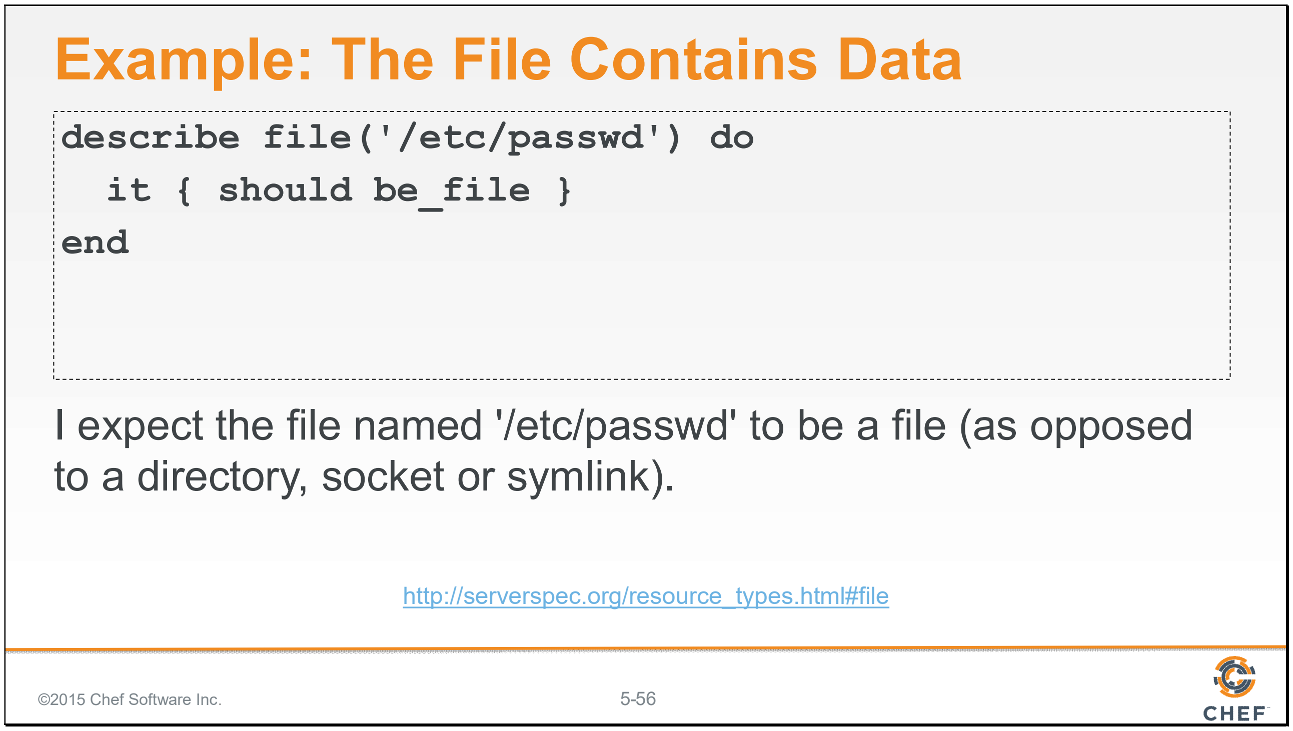
ServerSpec can help us assert different characteristics about files on the file system. Like if it is a file, directory, socket or symlink.

The file's mode owner or group. If the file is readable, writeable, or executable. It is even able to verify the data contained within the file.

http://serverspec.org/resource_types.html#file

CHEF

Slide 56



Example: The File Contains Data

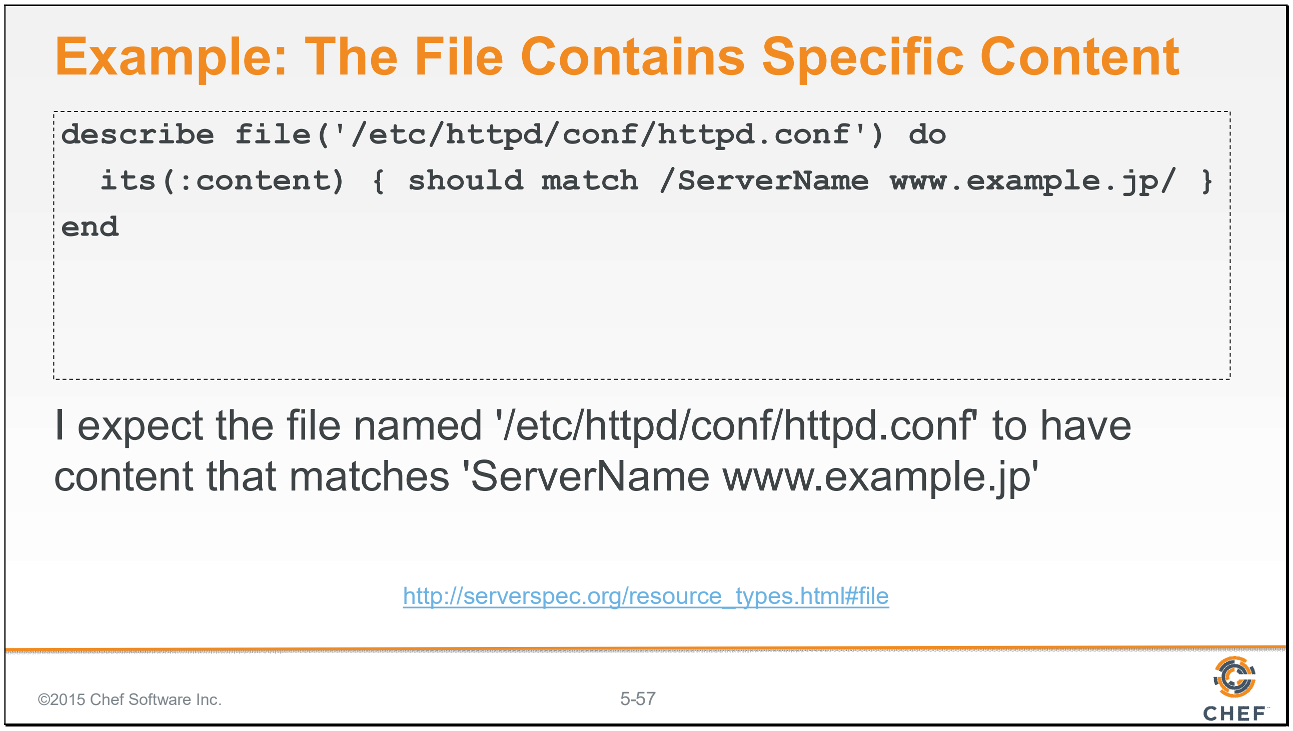
```
describe file('/etc/passwd') do
  it { should be_file }
end
```

I expect the file named '/etc/passwd' to be a file (as opposed to a directory, socket or symlink).

http://serverspec.org/resource_types.html#file

Here we are describing an expectation that the file named '/etc/passwd' is a file.

Slide 57



Example: The File Contains Specific Content

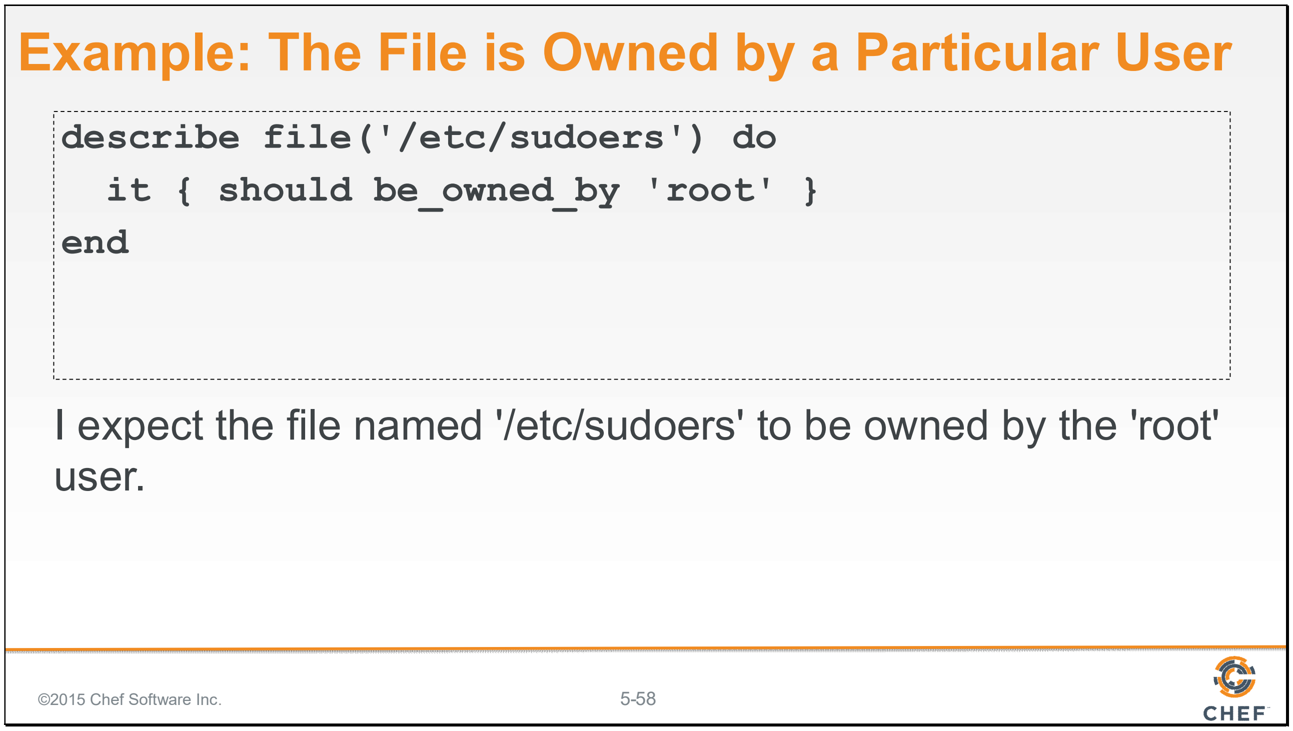
```
describe file('/etc/httpd/conf/httpd.conf') do
  its(:content) { should match /ServerName www.example.jp/ }
end
```

I expect the file named '/etc/httpd/conf/httpd.conf' to have content that matches 'ServerName www.example.jp'

http://serverspec.org/resource_types.html#file

Here we are describing an expectation that the file named '/etc/httpd/conf/httpd.conf' has contents that match the following regular expression. Asserting that somewhere in the file we will find the following bit of text.

Slide 58



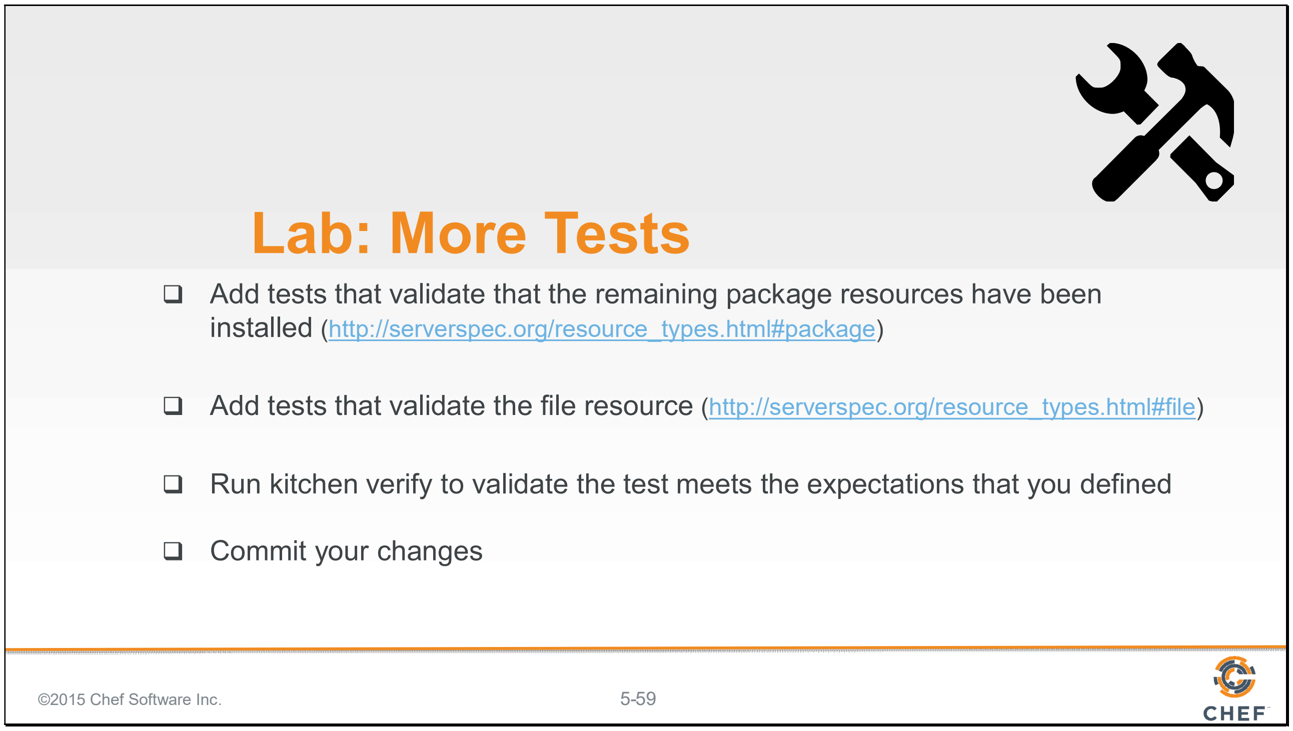
Example: The File is Owned by a Particular User

```
describe file('/etc/sudoers') do
  it { should be_owned_by 'root' }
end
```

I expect the file named '/etc/sudoers' to be owned by the 'root' user.

Here we are describing an expectation that the file named '/etc/sudoers' should be owned by the root user.

Slide 59



Lab: More Tests

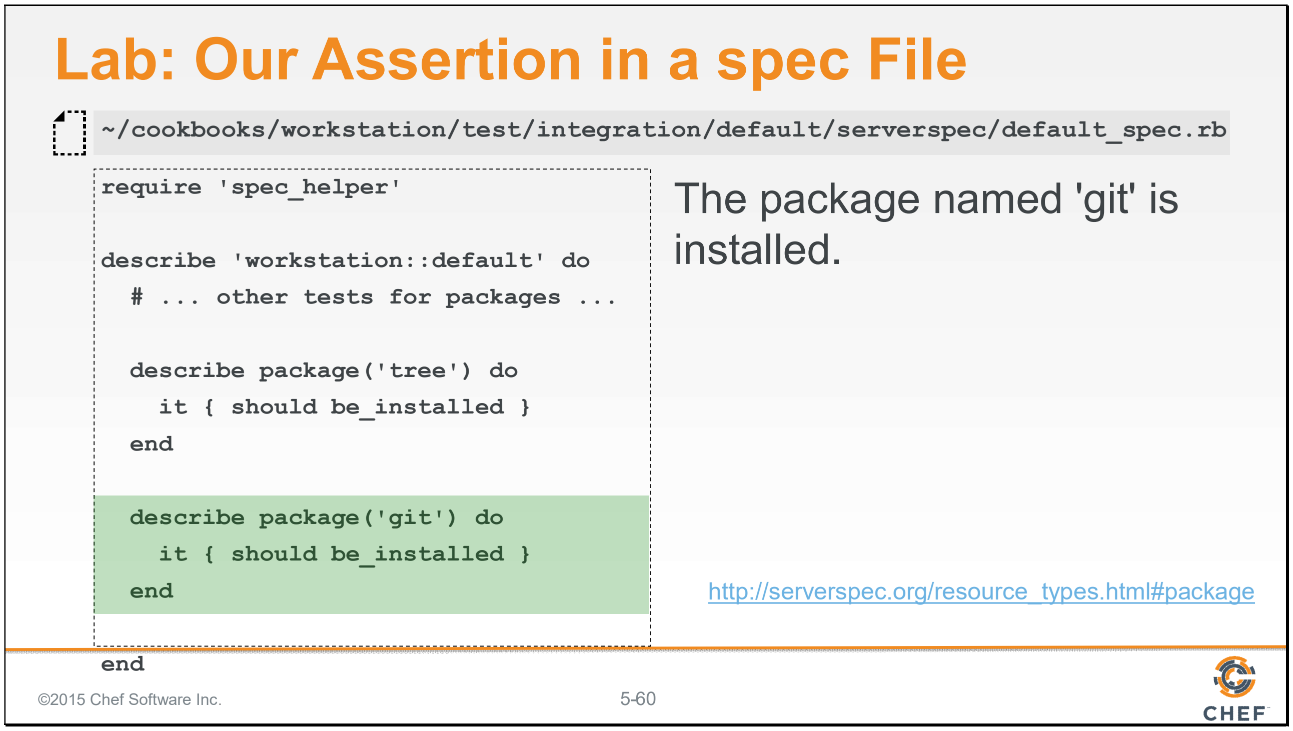
- ❑ Add tests that validate that the remaining package resources have been installed (http://serverspec.org/resource_types.html#package)
- ❑ Add tests that validate the file resource (http://serverspec.org/resource_types.html#file)
- ❑ Run kitchen verify to validate the test meets the expectations that you defined
- ❑ Commit your changes

As a lab exercise, we want you to define additional tests that validate the remaining resources within our default recipe.

Add tests for the remaining package resources that are converged by the "workstation" cookbook's default recipe.

You may also add tests for the file resource to ensure the file is present, that the contents are correctly defined, that it is owned by a particular user and owned by a particular group.

Slide 60



Lab: Our Assertion in a spec File

~/cookbooks/workstation/test/integration/default/serverspec/default_spec.rb

```
require 'spec_helper'

describe 'workstation::default' do
  # ... other tests for packages ...

  describe package('tree') do
    it { should be_installed }
  end

  describe package('git') do
    it { should be_installed }
  end

end
```

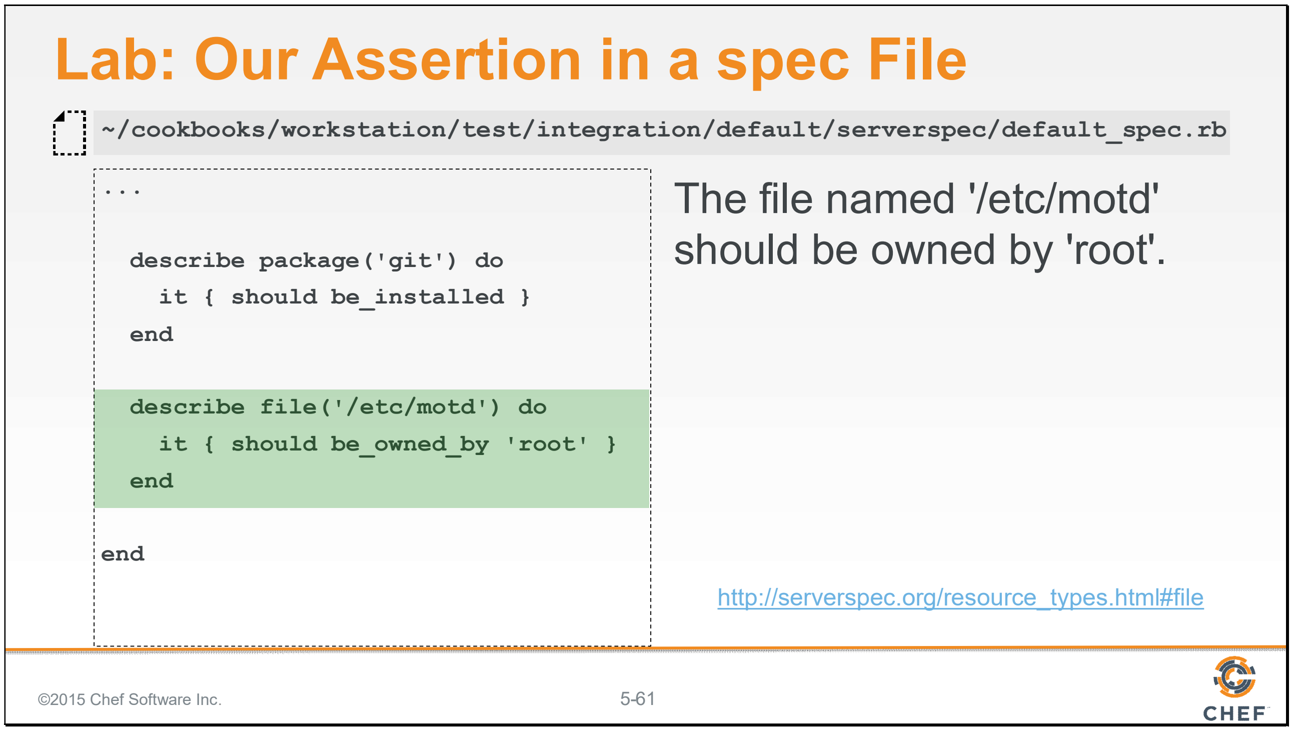
The package named 'git' is installed.

http://serverspec.org/resource_types.html#package

Let's review the lab.

Here we are verifying that the package git is installed. The structure of the test is very similar to the one we demonstrated earlier. You'll likely have another test that validates the editor you specified is also installed.

Slide 61



Lab: Our Assertion in a spec File

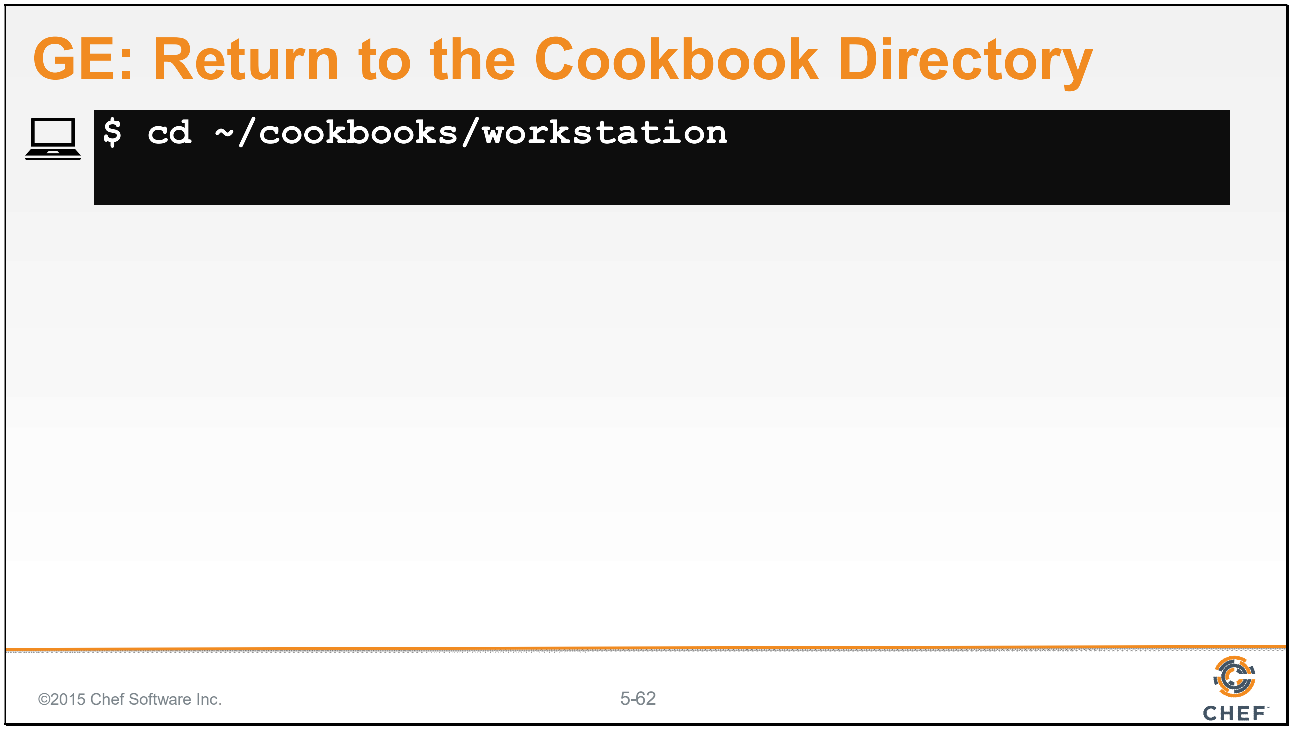
~/cookbooks/workstation/test/integration/default/serverspec/default_spec.rb

```
...

  describe package('git') do
    it { should be_installed }
  end

  describe file('/etc/motd') do
    it { should be_owned_by 'root' }
  end

end
```

The file named '/etc/motd' should be owned by 'root'.

http://serverspec.org/resource_types.html#file

For the file resource, we chose only to verify that the file named '/etc/motd' is owned by the root user. You may have verified that it was a file, that it belonged to a group, and that it contained content you felt important to verify.

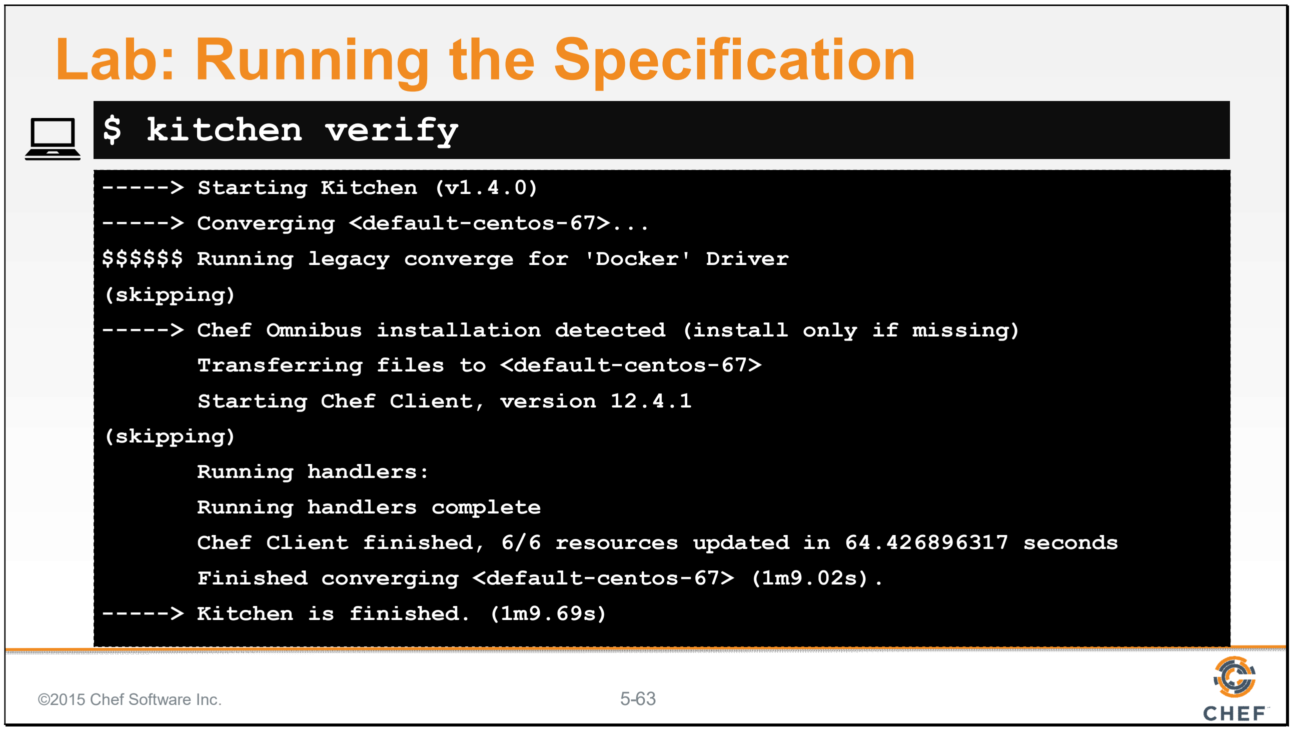


GE: Return to the Cookbook Directory

```
$ cd ~/cookbooks/workstation
```

Change into the workstation cookbook directory.

Slide 63



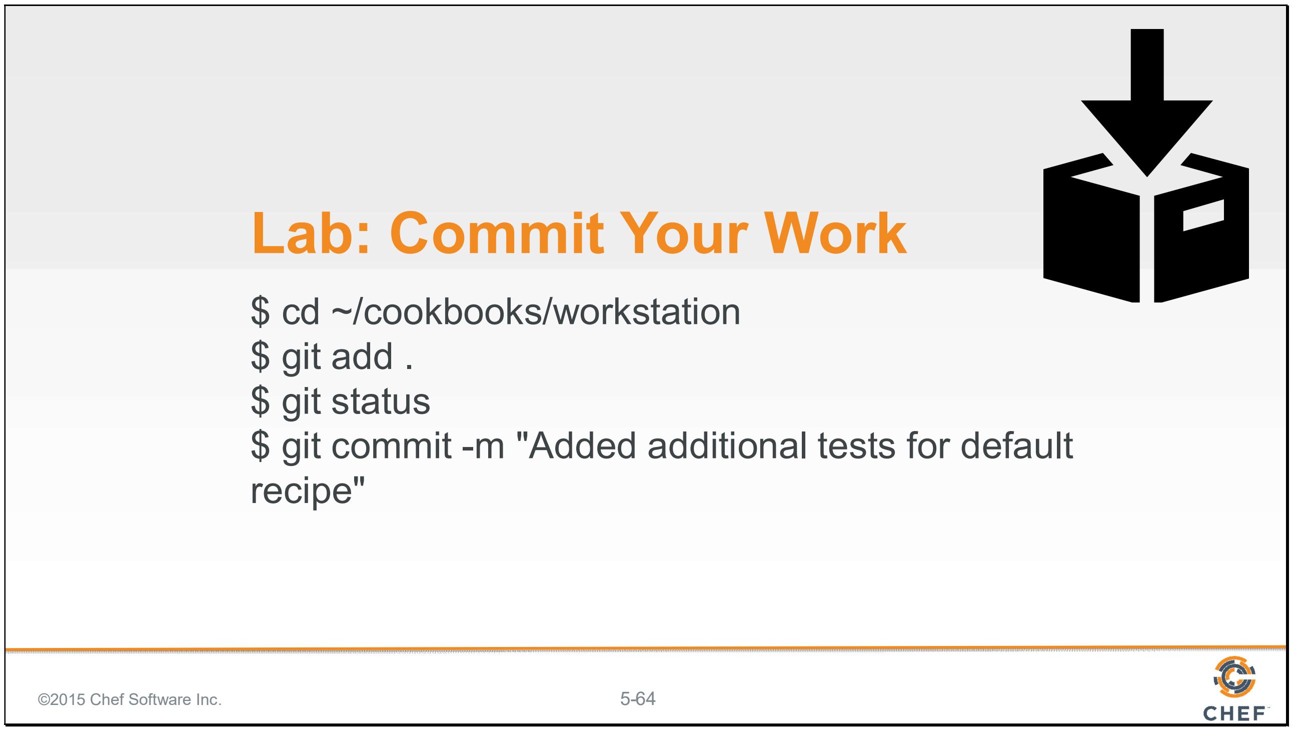
Lab: Running the Specification

```
$ kitchen verify
```

```
-----> Starting Kitchen (v1.4.0)
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
(skipping)
-----> Chef Omnibus installation detected (install only if missing)
       Transferring files to <default-centos-67>
       Starting Chef Client, version 12.4.1
(skipping)
       Running handlers:
       Running handlers complete
       Chef Client finished, 6/6 resources updated in 64.426896317 seconds
       Finished converging <default-centos-67> (1m9.02s).
-----> Kitchen is finished. (1m9.69s)
```

With more tests created lets verify all of these tests pass when we converged the workstation cookbooks default recipe. Use the `kitchen verify` command to execute the test

Slide 64

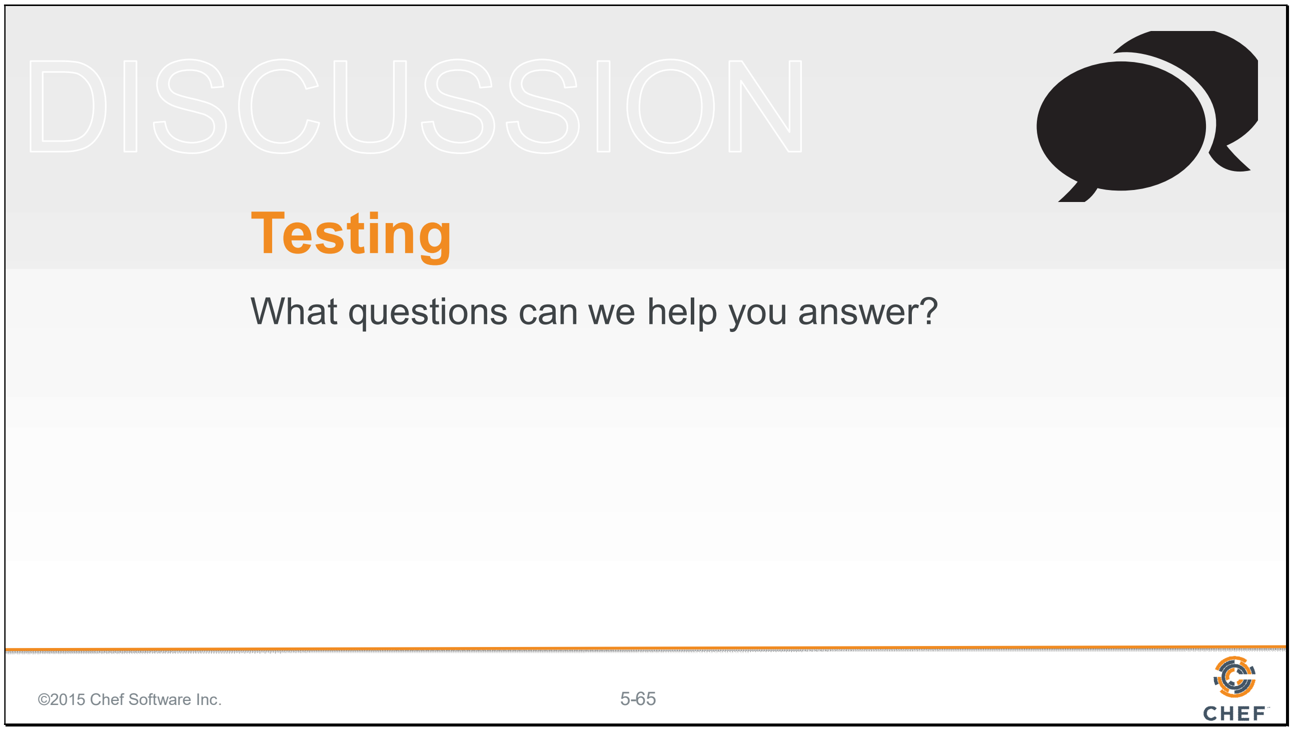


Lab: Commit Your Work

```
$ cd ~/cookbooks/workstation
$ git add .
$ git status
$ git commit -m "Added additional tests for default recipe"
```

If all the tests that you defined are working then it is time to commit our changes to version control.

Slide 65



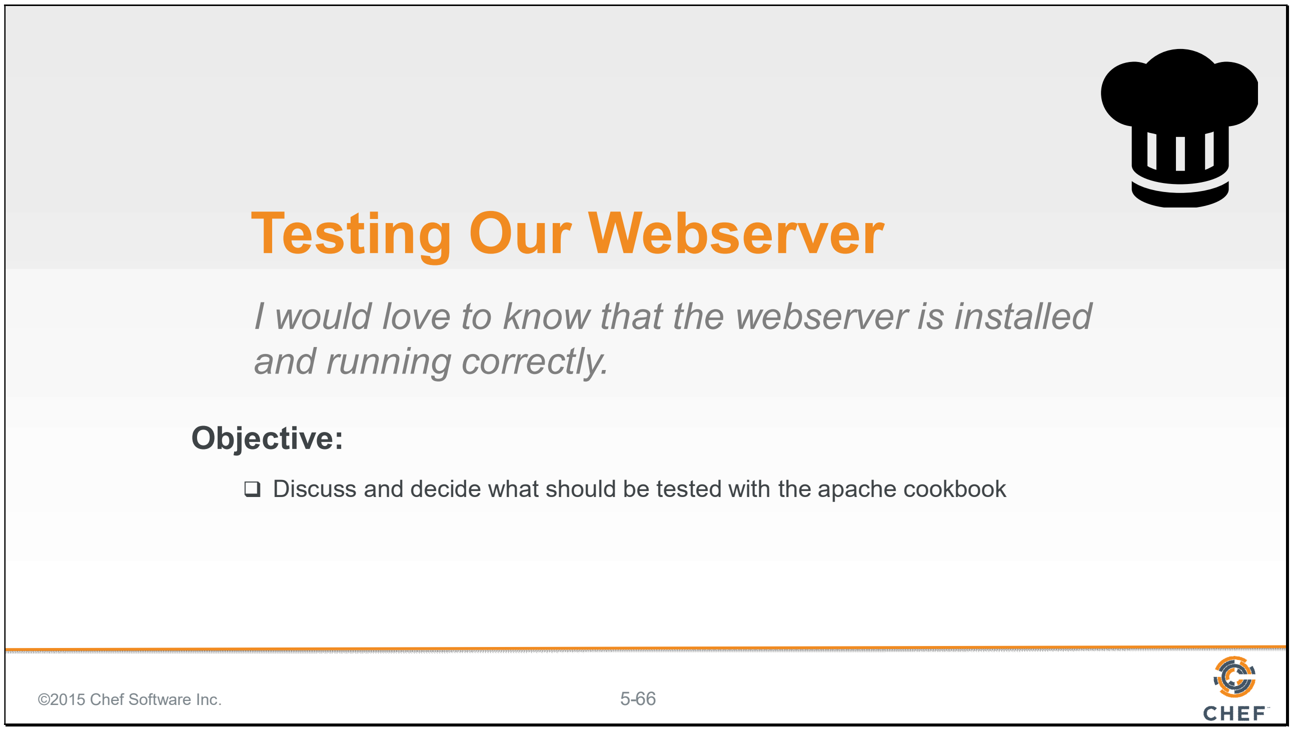
DISCUSSION

Testing

What questions can we help you answer?

What questions can we help you answer?

Slide 66



Testing Our Webserver

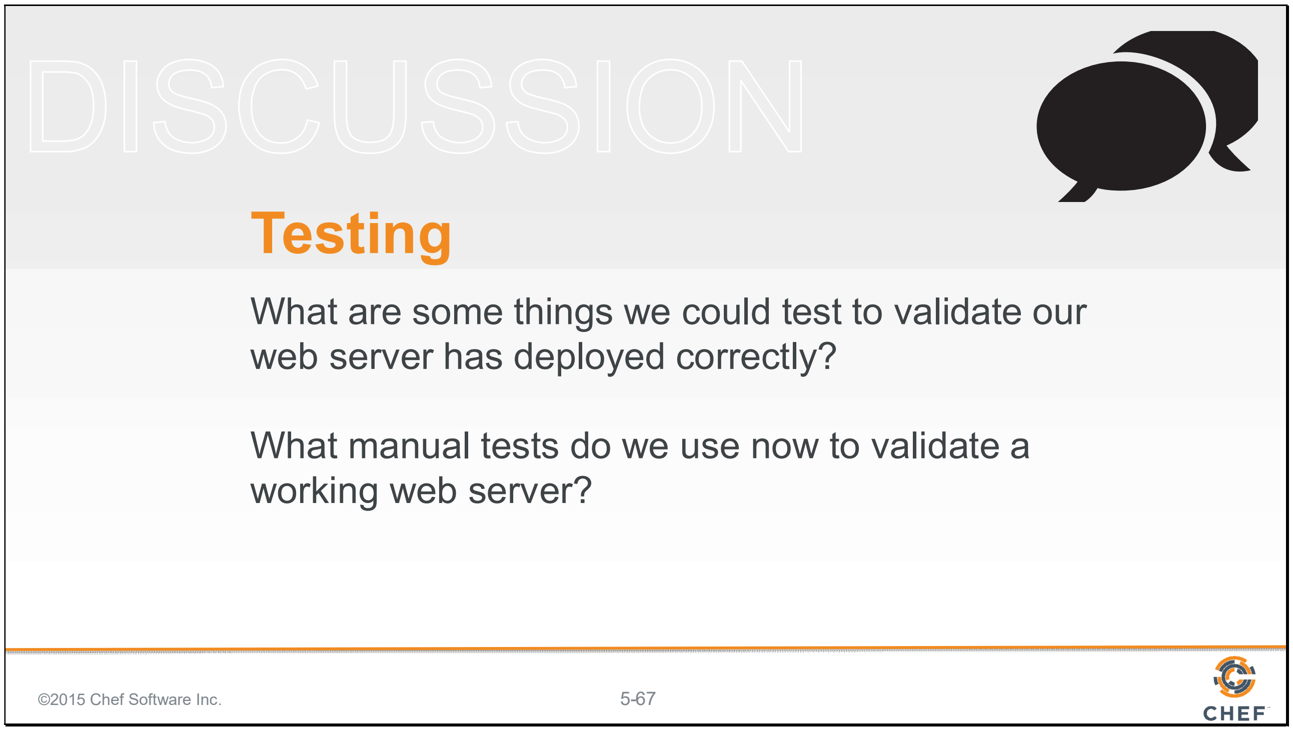
I would love to know that the webserver is installed and running correctly.

Objective:

- ❑ Discuss and decide what should be tested with the apache cookbook

Now let's turn our focus towards testing the apache cookbook.

Slide 67


DISCUSSION

Testing

What are some things we could test to validate our web server has deployed correctly?

What manual tests do we use now to validate a working web server?

What are some things we could test to validate our web server has deployed correctly?

The apache cookbook is similar to the workstation cookbook. It has a package and file which are things that we have already tested. The new thing is the service. We could review the ServerSpec documentation to find examples on how to test the service.

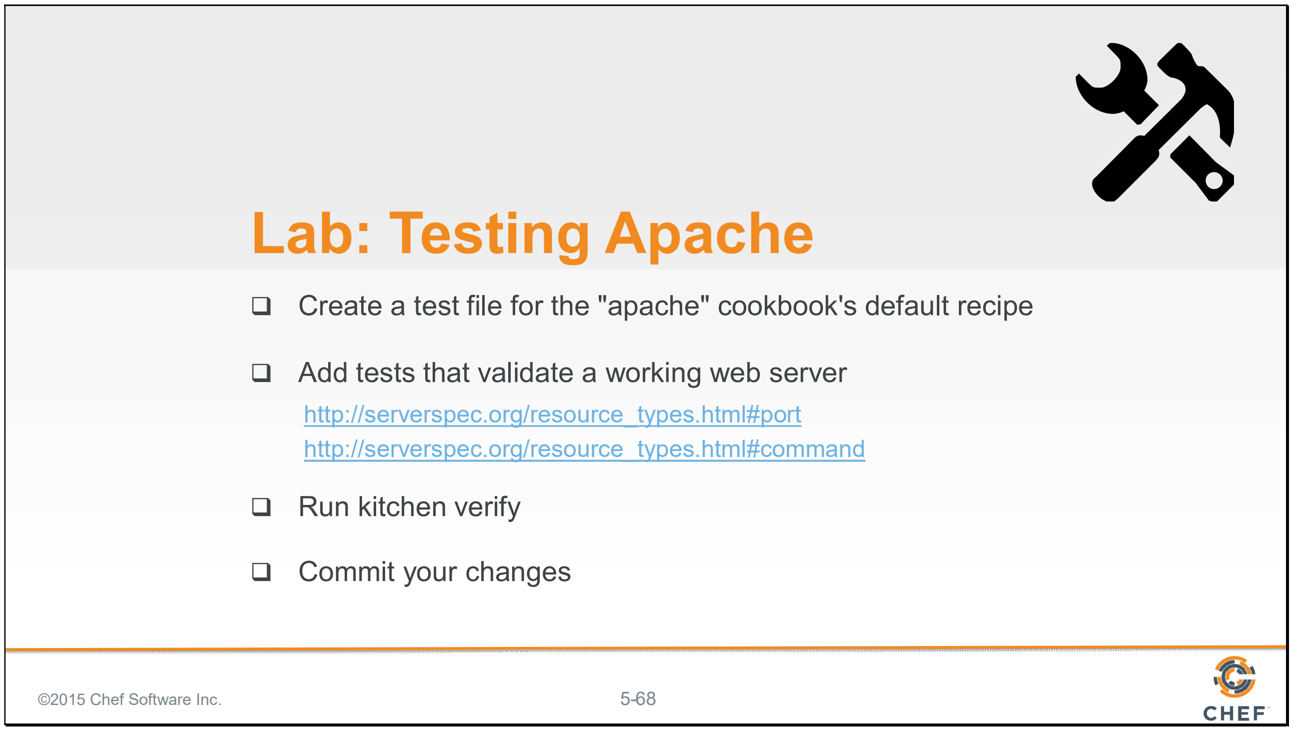
But does testing the package, file and service validate that apache is hosting our static web page and returning the content to visitors of the instance?

What manual tests do we use now to validate a working web server?

After applying the recipes in the past we visited the site through a browser or verified the content through running the command 'curl localhost'.

Is that something that we could test as well? Does ServerSpec provide the way for us to execute a command and verify the results?

Slide 68



Lab: Testing Apache

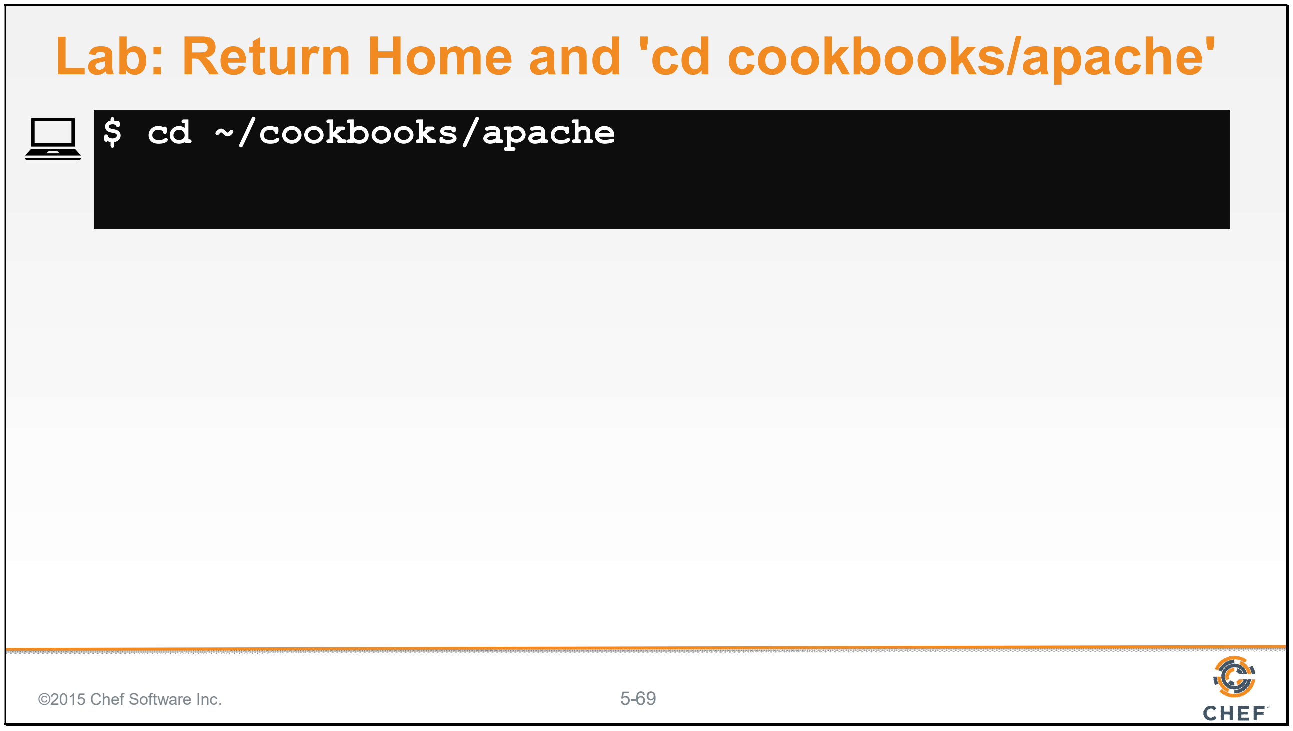
- ❑ Create a test file for the "apache" cookbook's default recipe
- ❑ Add tests that validate a working web server
  http://serverspec.org/resource_types.html#port
  http://serverspec.org/resource_types.html#command
- ❑ Run kitchen verify
- ❑ Commit your changes

So for this final exercise, you are going to create a test file for the apache cookbook's default recipe.

That test will validate that you have a working web server. This means I want you to add the tests that you feel are necessary to verify that the system is installed and working correctly.

When you are done execute your tests with `kitchen verify`.

Slide 69

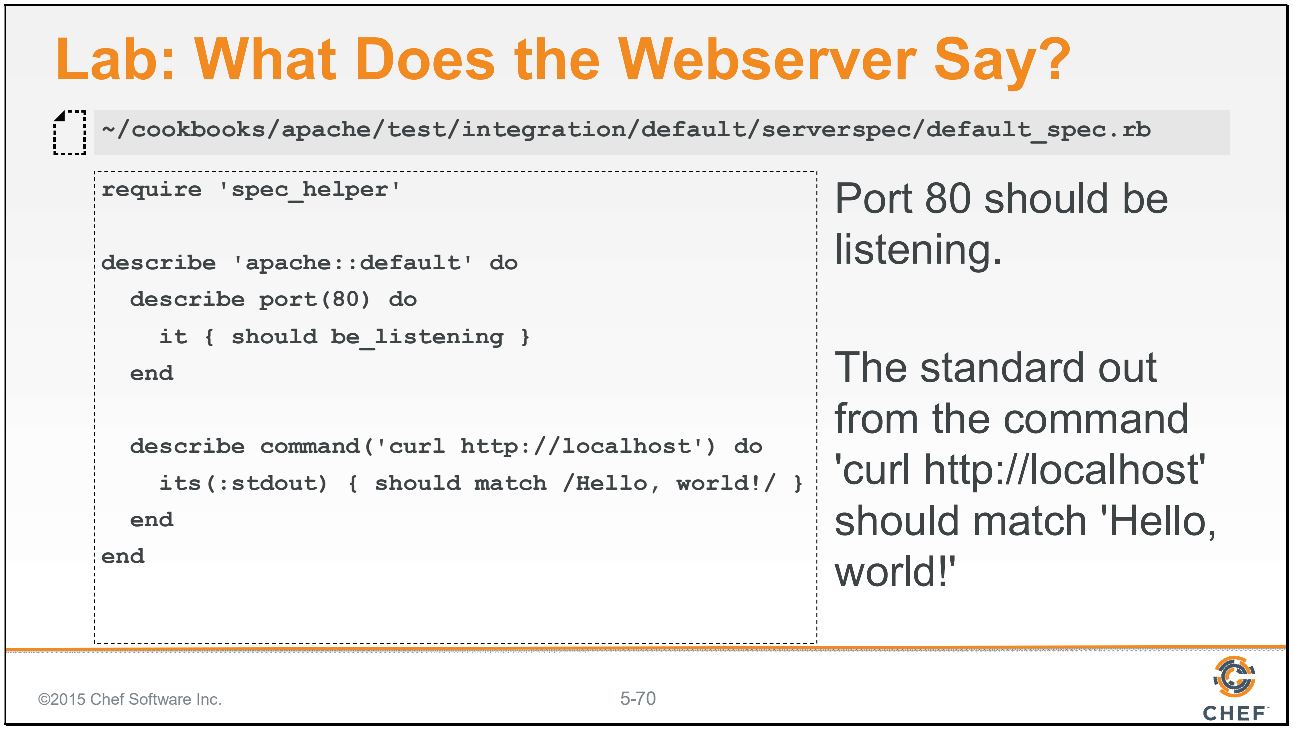


Lab: Return Home and 'cd cookbooks/apache'

```
$ cd ~/cookbooks/apache
```

Return home and then move into the apache cookbook's directory.

Slide 70



Lab: What Does the Webserver Say?

~/cookbooks/apache/test/integration/default/serverspec/default_spec.rb

```
require 'spec_helper'

describe 'apache::default' do
  describe port(80) do
    it { should be_listening }
  end

  describe command('curl http://localhost') do
    its(:stdout) { should match /Hello, world!/ }
  end
end
```

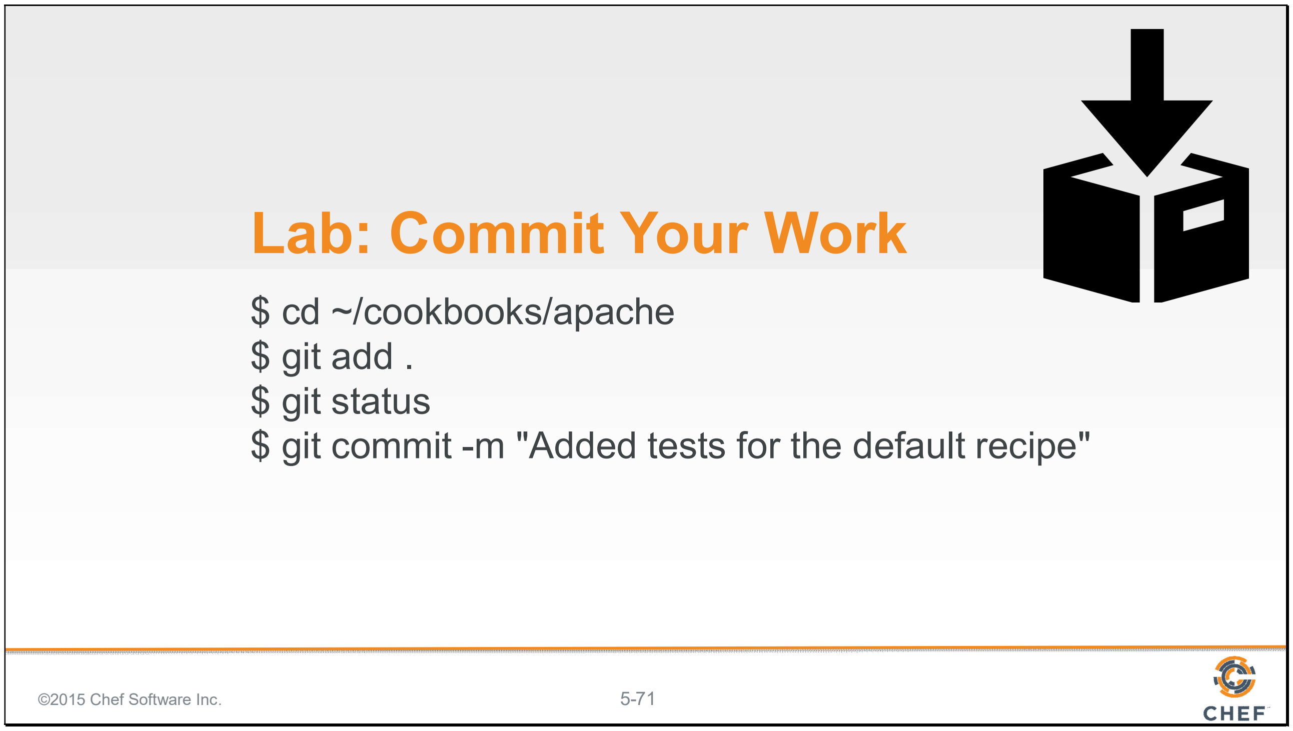
Port 80 should be listening.

The standard out from the command 'curl http://localhost' should match 'Hello, world!'

Here we chose to validate that port 80 should be listening for incoming connections.

And we also validated that the standard out from the command 'curl http://localhost' should match 'Hello, world!'.

Slide 71

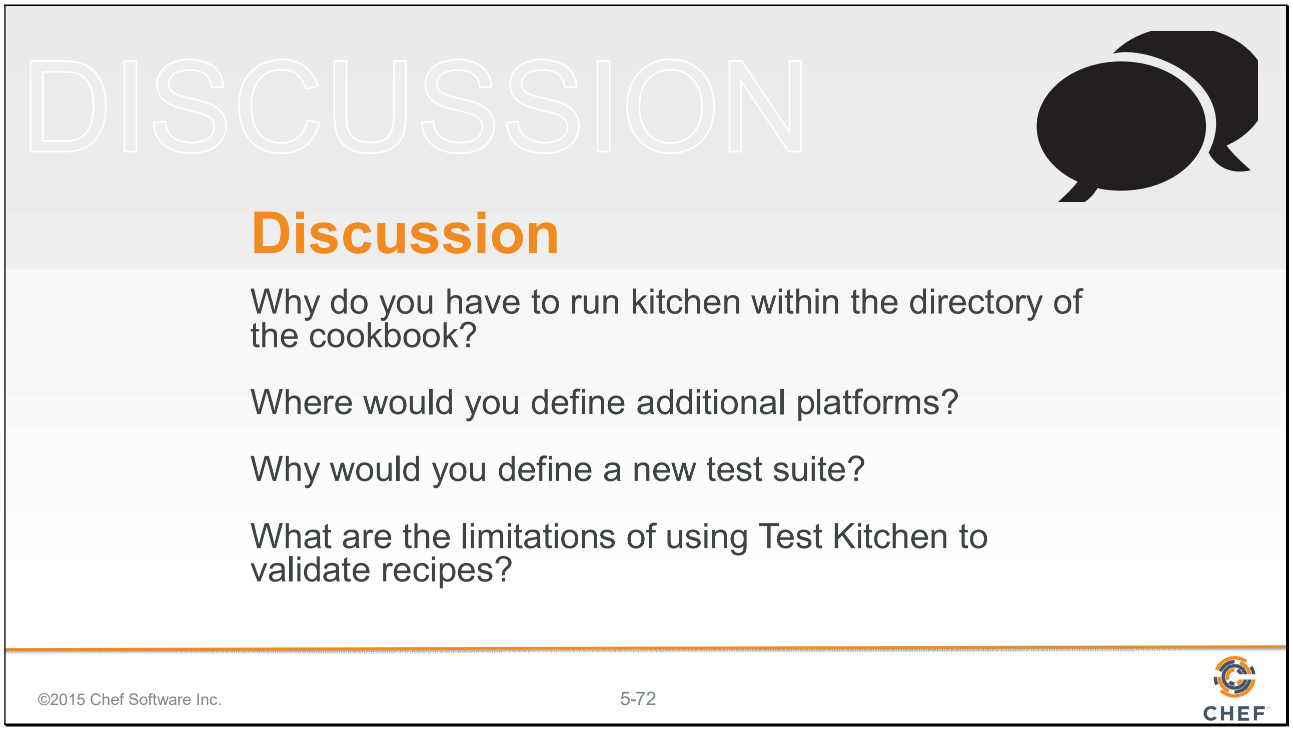


Lab: Commit Your Work

```
$ cd ~/cookbooks/apache
$ git add .
$ git status
$ git commit -m "Added tests for the default recipe"
```

Again, let’s commit the work.

Slide 72



DISCUSSION

Discussion

Why do you have to run kitchen within the directory of the cookbook?

Where would you define additional platforms?

Why would you define a new test suite?

What are the limitations of using Test Kitchen to validate recipes?

CHEF

Answer these questions.

With your answers, turn to another person and alternate asking each other asking these questions and sharing your answers.

Slide 73

DISCUSSION

Q&A

What questions can we help you answer?

- Test Kitchen
- kitchen commands
- kitchen configuration
- ServerSpec

What questions can we help you answer?

Generally or specifically about test kitchen, kitchen commands, kitchen configuration, ServerSpec.

Slide 74

CHEF™