



**CHEF**<sup>TM</sup>

Chef Training Services

# Chef Intermediate

Participant Guide



Welcome to Chef Intermediate

# Introduce Yourselves

Name

Current job role

Previous job roles / Background

Experience with Chef

Favorite Text Editor

Before we start let me introduce myself. Then I would like it if everyone had a chance to introduce themselves.

Instructor Note: Often times with larger, in-person groups I prefer to have the individuals perform this introduction one-on-one. Having people leave their desks and greet as many people as they can during the time allotted. I often feel this works better as it removes the pressure from the single individual to introduce themselves in a way that is presenting themselves and not actually greeting people. When Online, I create a pre-defined order, announce that order, and then invite a person to speak, thank them when they are done.

## Expectations

You will leave this class with the ability to extend the components of Cookbooks.

You bring with you your own domain expertise and problems. Chef is a framework for solving those problems. Our job is to teach you how to express solutions to your problems with Chef.

The goal of this training is to teach you techniques that will help you extend the functionality of your cookbooks. We also want to share the thought process on why and how to best employ these techniques.

Chef is built on top of Ruby. This means you have the power of a programming language at your disposal and we will have to keep a tight focus on the challenges and exercises presented in this content. During and throughout the content we will have discussion where we may have additional time to talk about many different topics but in this interest of time and popular opinion we may need to leave those discussions.

During the introductions you learned about the other individuals here in the course with you. They may have shared similar problems and domains. During the time that we are here respectfully reach out them so that you can continue the conversation, grow each others' knowledge, and become better professionals.

# Expectations

**Ask Me Anything:** It is important that we answer your questions and set you on the path to be able to find more answers.

**Break It:** If everything works the first time go back and make some changes. Explore! Discovering the boundaries will help you when you continue on your journey.

All throughout this training I strongly encourage you to ask questions whenever you do not understand a topic, an acronym, concept, or software. By asking a question you better your learning and often times better the learning of those with you in this training. Asking questions is a sign of curiosity that we want to encourage and foster while we are here together.

This curiosity can also be employed by exploring the boundaries of the tools you are using and the language you are writing. The exercises and the labs we will perform will often lead you through examples that work from the beginning to the end. When you develop solutions it is rare that something works from the start all the way to the end. Errors and issues come up from typos or the incorrect usage of a command of the programming language. When you fall off the path it can often be hard to find your way back. Here, if you find yourself always on the correct path explore what happens when you step off of it, what you see, the error messages you are presented with, the new results you might find.

# Group Exercises, Labs, and Discussion

This course is designed to be hands on. You will run lots of commands, write lots of code, and express your understanding.

- **Group Exercises:** All participants and the instructor will work through the content together. The instructor will often lead the way and explain things as we proceed.
- **Lab:** You will be asked to perform the task on your own or in groups.
- **Discussion:** As a group we will talk about the concepts introduced and the work that we have completed.

The content of this training has been designed in a way to emphasize this hands-on approach to the content. Together, we will perform exercises together that accomplish an understood objective. After that is done you will often emphasize an activity by performing a lab. The lab is designed to challenge your understanding and retention of the previously accomplished exercises. You can work through this labs on your own or in groups. After completing the labs we will all come together again to review the exercise. Finally, we will end each section with a discussion about the topics that we introduced. These discussions will often ask you to share your opinions, recent experiences, or previous experiences within this domain.

## Day 1

Introduction  
Why Write Tests? Why is that Hard?  
Writing a Test First  
Refactoring Cookbooks with Tests  
Faster Feedback with Unit Testing  
Testing Resources in Recipes  
Refactoring to Attributes  
Refactoring to Multiple Platforms

## Day 2

Approaches to Extending Resources  
Why Use Custom Resources  
Creating a Custom Resource  
Refining a Custom Resource  
Ohai  
Ohai Plugins  
Creating an Ohai Plugin  
Tuning Ohai

This is the outline of the events for this training. Please take a moment to review this list to ensure that the topics listed here meet your expectations. Take a moment to note which topics are of most interest to you. Also note which topics are not present here on this list. We will discuss your thoughts at the end of the section.

# EXERCISE



## Pre-built Workstation

*We will provide for you a workstation with all the tools installed.*

**Objective:**

- Login to the Remote Workstation

As I mentioned there is a lot work planned for the day. To ensure we focus on the concepts we introduce and not on troubleshooting systems we are providing you a workstation with the necessary tools installed to get started right away.

Instructor Note: At the end of the training it is often a good idea to offer your services to help individuals install necessary software or troubleshoot their systems.

## Login to the Workstation



```
> ssh IPADDRESS -l USERNAME
```

```
The authenticity of host '54.209.164.144 (54.209.164.144)' can't
be established. RSA key fingerprint is
SHA256:tKoTsPbn6ER9BLThZqntXTxIYem3zV/iTQWvhLrBIBQ. Are you sure
you want to continue connecting (yes/no)? yes
chef@54.209.164.144's password: PASSWORD
chef@ip-172-31-15-97 ~]$
```

I will provide you with the address, username and password of the workstation. With that information you will need to use the SSH tool that you have installed to connect that workstation.

This demonstrates how you might connect to the remote machine using your terminal or command-prompt if you have access to the application ssh. This may be different based on your operating system.

# EXERCISE



## Pre-built Workstation

*We will provide for you a workstation with all the tools installed.*

**Objective:**

- ✓ Login to the Remote Workstation

Now that you are connected to that workstation we have taken care of all the necessary work to get started with the training.

# DISCUSSION



## Discussion

What topics are you most interested in learning?

What topics are missing that you want to learn about?

Let us end with a discussion about the following topics.

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?

Before we continue let us stop for a moment answer any questions that anyone might have at this time.



## Why Write Tests? Why is that Hard?

©2018 Chef Software Inc.

2-1



Why should you write tests? Why is important that we write tests for the recipes and the cookbooks that we define. Some of you here may be because you are starting to see an importance to what testing can provide. Others of you may not be convinced. Wherever you stand, the real reason you came here to learn is to break down the barriers that make testing hard. Because testing is hard!

# EXERCISE



## Why Write Tests? Why is that Hard?

*Should I write a test? Perhaps the answer to that question lies in: why write tests?*

**Objective:**

- Discussion about Writing Tests
- Discussion about Why Writing Tests is Hard

All of you likely have a personal answer or opinions to these questions. Good. Capture those because we will have a discussion together. To start the discussion I will provide my thoughts and opinions about why I think it is important to write tests. Then I want you to share your thoughts. Then we will discuss the many reasons that testing is hard.

# CONCEPT

## Current Cookbook Development Workflow



1. Write some cookbook code
2. Perform ad-hoc verification
3. Upload the cookbook to the Chef Server

©2018 Chef Software Inc. 2-3 

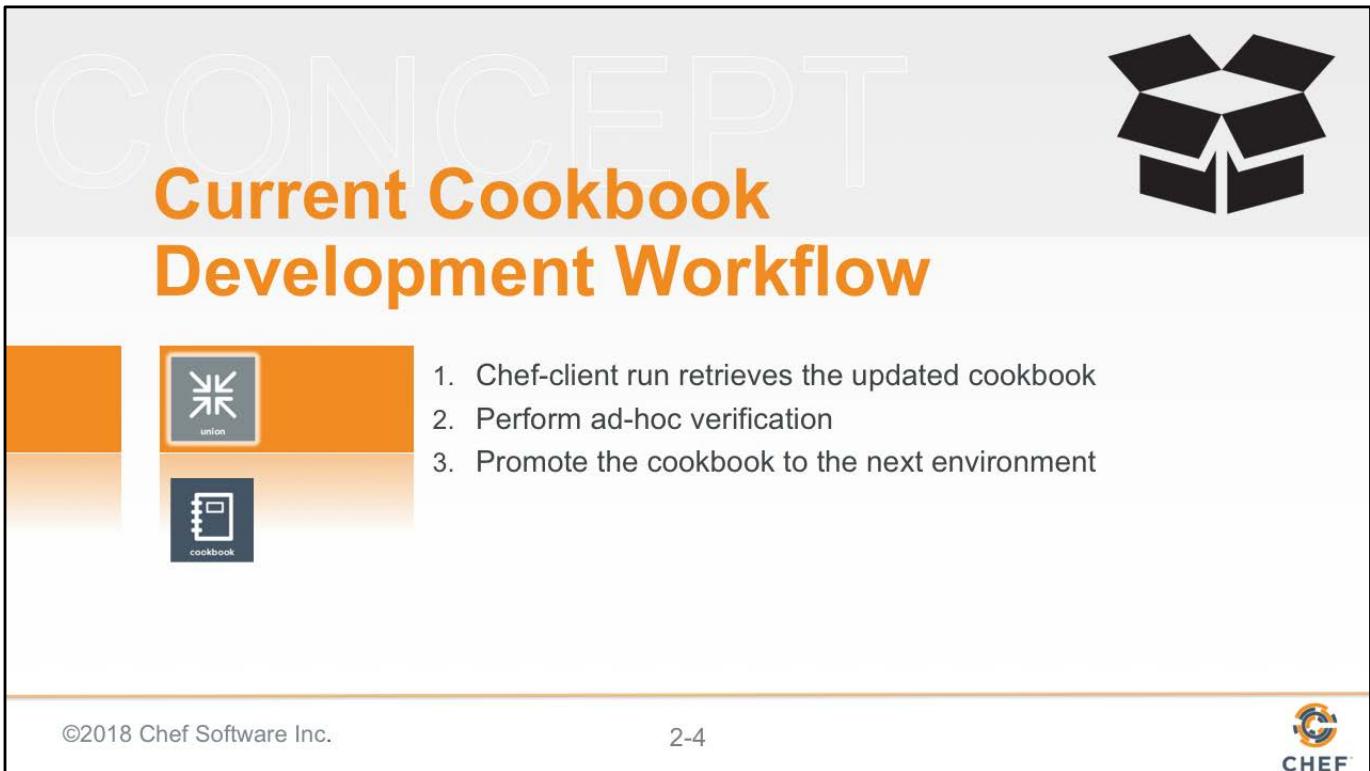
To understand why it is important to write tests I believe it is important to examine the current cookbook development workflow that most individuals employ.

To provide a few answers to why writing tests are powerful and why are they hard to write we need to look at our current cookbook development workflow.

On your local workstation you will write cookbook code. Creating a new recipe to meet new requirements, fixing a bug in an existing recipe, or refactoring complicated recipes into several smaller recipes, helper methods, or maybe even a custom resource.

When you are done with those changes you will spend a few moments visually scanning the code to ensure that your syntax is correct. That every block you start with a 'do' has a matching 'end'. Check your node attributes for spelling issues. Each key-value pair within the hash has a comma that follows.

After enough examination we feel comfortable to upload the cookbook to the Chef Server.



The slide features a large, semi-transparent watermark-like text "CONCEPT" at the top left. In the top right corner is a black icon of an open box. Below the watermark, the title "Current Cookbook Development Workflow" is displayed in large, bold, orange font. To the left of the title is a vertical stack of three colored squares: orange at the top, followed by a gradient of orange and yellow, and finally dark blue/black at the bottom. On the orange square, there is a white icon of a stylized 'K' with the word "union" below it. On the dark blue/black square, there is a white icon of a book with the word "cookbook" below it. To the right of the title, a numbered list of three steps is shown:

1. Chef-client run retrieves the updated cookbook
2. Perform ad-hoc verification
3. Promote the cookbook to the next environment

At the bottom left, the copyright notice "©2018 Chef Software Inc." is present. At the bottom center, the page number "2-4" is displayed. In the bottom right corner is the official Chef logo, which consists of a circular icon with the letter "C" inside, followed by the word "CHEF".

You login to a test node that you patiently bootstrap into a union environment. This is an environment we setup with no cookbook restrictions allowing chef-client to synchronize and apply the latest changes in the recently completed cookbook. Here you see if you got the right package names, spelled all our cookbook attributes correctly, and didn't typo any of the configuration in the templates. If everything converges without error you poke around the system -- running a few commands to see if ports are blocked, services are running, and the logs don't show any errors. Logging out of the working system you feel pretty comfortable promoting the cookbook to the rehearsal environment.

# CONCEPT

## Current Cookbook Development Workflow



1. Chef-client run retrieves the updated cookbook
2. Perform ad-hoc verification
3. Establish monitoring for deployed services

©2018 Chef Software Inc. 2-5 

Here in this new environment you may log into another system. Manually perform a chef-client run and then poke around again if everything works. You also may not. It was such a small change and everything worked on the other machine -- so it's likely to work here. Right? Instead of running through a series of ad-hoc verifications again on a new system in this environment - you start to think of the backlog of things that need to get done.

# PROBLEM



## Risk

Every change to our cookbooks introduce risk. Validating every change would take too long in this system. To alleviate that we often batch these changes up. Batching up the changes make it harder to discover when we introduced an error.

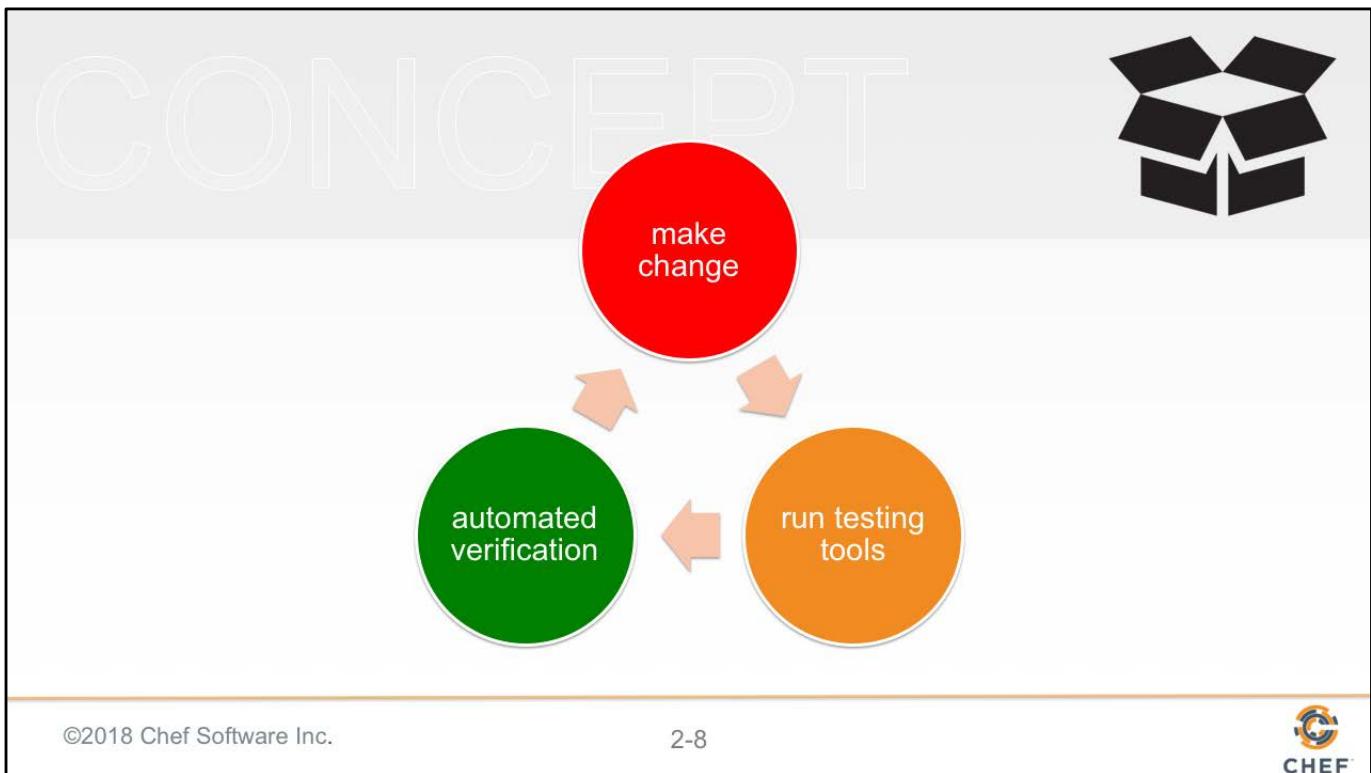
Every time we make changes to our cookbooks we are introducing risk. Ideally we would validate every change to the cookbook but often do not because the amount of time it takes is far too prohibitive. Instead we often will batch up these changes into a set that we will validate. A set of changes like this can often hide errors that we may have introduced. This is definitely true as the complexity of the cookbook code increases.

We have a choice. We can slow down; validating every change. We can also stop making changes altogether. Or we can adopt new practices, like testing, to help us validate these changes faster; allowing us to continue to move quickly as we continue to satisfy new requirements.



Carrying out testing at every stage (e.g. union, rehearsal) gives great feedback on its success at the cost of the time required for each cookbook to be pushed through this workflow.

Every change needs to be verified in this manner because Ruby, the language Chef is built on, is a dynamically typed programming language. Dynamically typed languages do checking at run-time as opposed to compile-time. This means that ruby files in our cookbook are not executed, thus not validated, until they are run. We also have the problem that we may even write the Ruby correctly but fail to understand the state of the host Operating System (OS) we are attempting to deploy against.



Writing and executing tests decreases the amount of time spent between when you make a change to when you can verify that change. This reduces the risk within the system.

How testing does address the speed of execution is by removing many of the outside dependencies and allowing you to execute your recipes against in-memory representations of the environment. Or automating the management of virtual machines and the process of executing your recipes against those virtual machines. And second, by allowing you to capture and automate the work that was previously performed in ad hoc verification.

# DISCUSSION



## Discussion

What are reasons you see for writing tests?

What are reasons you see for not writing tests?

I shared with you my opinion on why I think it is important to write tests. Now I would like to understand what reasons you see for writing tests. I would also like to know your reasons for not writing tests.

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.

# EXERCISE



## Why Write Tests? Why is that Hard?

*To test or not to tests. It's no longer the question. Now I need to think: what makes writing tests challenging?*

**Objective:**

- Discussion about Writing Tests
- Discussion about Why Writing Tests is Hard

You may or may not be convinced that there is value in writing and executing tests. If the opinions we all have expressed has not convinced you I encourage you to continue to find more discussions where you can hear more opinions and share yours with others. It is important to have these discussions within your teams and your organization.

I want to now focus the discussion on the reasons why writing tests are hard. Similar to the previous discussion I want to provide my opinion to start the discussion. I want you to also contribute your opinions and experiences as they are equally valuable.

# CONCEPT



## Another Language

Learning to write tests require you to learn a whole new language that you must understand grammatically.

The language you use to define your tests in is not the same as the language you use to compose your original intentions. To test your code you need to write more code. However, this new code that you write is different as you are expressing your desired expectations of the system across a number of scenarios. This requires you to learn one or more new languages which have completely new systems and structures.

Testing asks you to solve a different problem in a different order when compared to process of writing software. You have to overcome particular challenges created by an implementation and express the desired expectations of that implementation before it is even built.

# CONCEPT



## Another Workflow

Executing tests requires learning new tools, commands, flags and configurations with entirely new mechanisms that provide you feedback.

Testing also asks you to change your behaviors through the new tools required to execute the tests. These tools represent a huge domain of knowledge expressed in all the commands, flags, and configuration that must be understood to be used correctly and then effectively as the complexity of your testing tools grow. The largest, and most immediate impact is on your development workflow which has to adopt new steps that feel unsure and even more unreliable as you receive a barrage of feedback in unfamiliar formats.

# DISCUSSION



## Discussion

What are reasons you see that make testing hard?

What are some of the ways in which you have made it less hard?

I shared with you my opinion on why I think it is hard to write tests. Now I would like to understand what reasons you see that make testing hard.

After we have expressed a set of reasons we should leave time within the discussion to discuss ways in which you have made it less hard.

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.

# EXERCISE



## Why Write Tests? Why is that Hard?

*I may or may not be convinced. The important thing is I understand what others around me think ... now I have more information to make up my mind.*

### Objective:

- ✓ Discussion about Writing Tests
- ✓ Discussion about Why Writing Tests is Hard

With our two discussions complete lets pause now for any questions that were not covered or even came out of the discussions.

# DISCUSSION



## Q&A

What questions can we answer for you?

Before we complete this section and start learning some of these new tools and languages let us pause for questions.



## Writing a Test First

Writing tests are often difficult. Writing tests before you have written the code that you want to test can often feel like a leap of faith. An act that requires a level of clairvoyance reserved for magicians or con-artists. Some have likened it towards starting a story by first writing the conclusion.

# CONCEPT



## Test Driven Development

1. Define a test set for the unit first
2. Then implement the unit
3. Finally verify that the implementation of the unit makes the tests succeed.
4. Refactor

Test Driven Development (TDD) is a workflow that asks you to perform that act continually and repeatedly as you satisfy the requirements of the work you have chosen to perform.

TDD generically focuses on the unit of software at any level. It is the process of writing the test first, implementing the unit, and then verifying the implementation with the test that was written.

A 'unit' of software is purposefully vague. This 'unit' is definable by the individuals developing the software. So the size of a 'unit of software' likely has different meanings to different individuals based on our backgrounds and experiences.

# CONCEPT



## Behavior Driven Development (BDD)

Behavior-driven development (BDD) specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit.

Borrowing from [agile software development](#) the "desired behavior" in this case consists of the requirements set by the business — that is, the desired behavior that has [business value](#) for whatever entity commissioned the software unit under construction.

Within BDD practice, this is referred to as BDD being an "outside-in" activity.

How you choose to express the requirements of that unit is the crux of Behavior Driven Development (BDD). Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit. Expressing this desired behavior is often expressed in scenarios that are written in a Domain Specific Language (DSL).

The cookbooks and recipes that you have written so far share quite a few similarities with BDD. In Chef, you express the desired state of the system through a DSL, resources, you define in recipes.

# CONCEPT



## TDD and BDD

**TDD** is a workflow process.

**BDD** influences the language we use to write tests and how we focus on the tests that matter.

TDD is a workflow process: Add a test; Run the test expecting failure; Add code; Run the test expecting success. Refactor.

BDD influences the language we use to write the tests and how we focus on tests that matter. The activities within this module focus on the process of taking requirements, expressing them as expectations, choosing one implementation to meet these expectations, and then verifying we have met these expectations.

# Objectives

After completing this module, you should be able to:

- Write an integration test
- Use Test Kitchen to create, converge, and verify a recipe
- Develop a cookbook with a test-driven approach

In this module you will learn how to use chef to generate a cookbook, write an integration test first, use Test Kitchen to execute that test, and then implement a solution to make that test pass.

## Building a Web Server

1. Install the httpd package
2. Write out a test page
3. Start and enable the httpd service

To explore the concepts of Test Driven Development through Behavior Driven Design we are going to focus on creating a cookbook that starts with the goal that installs, configures, and starts a web server that hosts the your company's future home page.

This cookbook will start very straight-forward and over the course of these modules we will introduce new requirements that will increase its complexity.

The goal again is to focus on the TDD workflow and understanding how to apply BDD when defining these tests. We are not concerned about focusing on best practices for managing web servers or modeling a more initially complex cookbook.

## Defining Scenarios

**Given SOME CONDITIONS**

**When an EVENT OCCURS**

**Then I should EXPECT THIS RESULT**

When requirements come to us it is rare that the product owners and customers ask us to deliver a particular technology or a software. In our case, I have asked you to setup a web page for your company. I did not specifically state a particular technology but to help limit the scope I have chosen that we are going to build this initial website with Apache.

Behavior driven design asks us to look at the work that we perform from the perspective of our users. Our first job is to develop the scenario that validates the work that we are about to accomplish.

These scenarios that we write are often written in the following format.

This very generically defines any scenario. What we need to do is apply this scenario format to our requirements.

## The Why Stack?

You should discuss...the feature and [pop the why stack](#) max 5 times (ask why recursively) until you end up with one of the following business values:

- Protect revenue
- Increase revenue
- Manage cost

If you're about to implement a feature that doesn't support one of those values, chances are you're about to implement a non-valuable feature. Consider tossing it altogether or pushing it down in your backlog.

- Aslak Hellesøy, creator of Cucumber

If our goal is to setup a new webpage we need to start to ask ourselves the question: Why. Why do we need to setup a website? Asking this question will help us identify for who the website is for and what purpose does it serve for the actor in this scenario.

Often times the why will raise more questions which you continue to ask why. You should do that. Asking why enough times will lead you to the true reason why you are taking action. The interesting thing is that knowing the true reason why will help reinforce your course of action or maybe change it entirely.

## Scenario: Potential User Visits Website

Given that I am a potential user

When I visit the company website in my browser

Then I should see a welcome message

The typical reason for setting up a website is to allow customers, users, potential users to learn more about the company. The needs of the website may change in the future but the first minimum viable product (MVP) is to simply give our users the ability to find out more information.

Our goal now is to define a scenario with this understanding.

This first scenario is enough information to help us build this cookbook with a TDD approach. This practice of defining a scenario is a tactic that I employ to help focus me on the most valuable work that needs to be done.

Important things to notice in the following scenario is the distinct lack of technology or implementation. The scenario is not concerned about the services that are running or files that might be found on the file system.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

### Objective:

- Examine the cookbook
- Write tests that verifies the cookbook does what we want it to do
- Execute the tests and see failure
- Write the recipe to make the test pass
- Execute the tests and see success

With the scenario defined it is now time for us to develop the cookbook. We are going to move through the following steps together to accomplish this task.

## Let's Start this Journey in the Home Directory



```
> cd ~
```

Let's start the journey on your workstation. From the home directory we are going to creating this cookbook.

## View the Tests in the Generated Cookbook



```
> tree apache
```

```
apache/
├── Berksfile
├── chefignore
├── LICENSE
├── metadata.rb
├── README.md
└── recipes
    └── default.rb
    ...
7 directories, 9 files
```

We can examine the contents of the cookbook that chef generated for us. Here you see that the tool created for us a complete test directory structure.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Examine the cookbook
- Write tests that verifies the cookbook does what we want it to do
- Execute the tests and see failure
- Write the recipe to make the test pass
- Execute the tests and see success

With the cookbook created it is now time to write that first test that verifies the cookbook does what we want it to do.

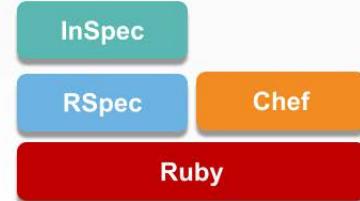
# CONCEPT



## RSpec and InSpec

RSpec is a Domain Specific Language (DSL) that allows you to express and execute expectations. These expectations are expressed in examples that are asserted in different example groups.

InSpec provides helpers and tools that allow you to express expectations about the state of infrastructure.



RSpec is a Behavior Driven Development (BDD) framework that uses a natural language domain-specific language (DSL) to quickly describe scenarios in which systems are being tested. RSpec allows you to setup a scenario, execute the scenario, and then define expectations on the results. These expectations are expressed in examples that are asserted in different example groups.

RSpec by itself grants us the framework, language, and tools. InSpec provides the knowledge about expressing expectations about the state of infrastructure.

## Auto-generated Spec File in Cookbook

```
~/apache/test/smoke/default/default_test.rb

unless os.windows?

  # This is an example test, replace with your own test.
  describe user('root'), :skip do
    it { should exist }
  end
end

# This is an example test, replace it with your own test.
describe port(80), :skip do
  it { should_not be_listening }
end
```

The generator created an example specification (or spec) file. Before we talk about the RSpec/InSpec language lets explain the long file path and its importance.

# CONCEPT



## Where do Tests Live?

```
~/apache/test/smoke/default/default_test.rb
```

Test Kitchen will look for tests to run under this directory. This corresponds to the value specified in the Test Kitchen configuration file (.kitchen.yml) in the suites section.

Let's take a moment to describe the reason behind this directory path. Within our cookbook we define a test directory and within that test directory we define another directory named 'smoke'. This is the basic file path that Test Kitchen expects to find the specifications defined in InSpec. The next part the path, 'smoke', corresponds to the path specified in the .kitchen.yml file.

# CONCEPT



## Where do Tests Live?

```
~/apache/test/smoke/default/default_test.rb
```

The default\_test.rb file is a Ruby file that contains the tests that we want to run when we spin up a test instance.

The ruby file, default\_test.rb, contains the tests that we have defined. A test file is a Ruby file that contains domain specific language constructs that we use to express our desired state of the system.

Let's open this default\_test.rb file and review the contents of it.

## Components of a InSpec Example

```
unless os.windows?  
  describe user('root'), :skip do  
    it { should exist }  
  end  
end
```

OS conditional

InSpec resource

expectation

When not on Windows, I expect the user named 'root', to exist.

The outermost statement is a conditional that states that when we want to evaluate the contents in between when we are not on Windows (e.g. CentOS, Ubuntu, Debian).

The inner describe has two parameters: The first is the the user resource named 'root' on the test instance. The second is the block which contains the expectations that we want to assert for the given resource.

Within the block we can define any number of expectations about the particular resource in the description. In this instance we are saying that we expect the user, named 'root', to exist on the instance. After the expectation that has been defined is a skip. This skip is a reminder that the examples have been defined in this test file were automatically generated and should be updated or removed.

## Components of a InSpec Example

```
describe port(80), :skip do
  it { should_not be_listening }
end
```

InSpec resource

expectation

When on any platform, I expect the port 80 **not** to be listening for incoming connections.

The second example within the test file describes the port 80 on any operating system and states the expectation that it does not expect port 80 to be listening.

By default all operating systems will be examined. So this example would be evaluated and executed against every operating system.

## Remove the Test for the root User

```
~/apache/test/smoke/default/default_test.rb

unless os.windows?

  # This is an example test, replace with your own test.
  describe user('root'), :skip do
    it { should exist }
  end
end

# This is an example test, replace it with your own test.
describe port(80), :skip do
  it { should_not be_listening }
end
```

Within the test file found at the following path you will find that it is already populated with that initial code. Remove the first test that asserts that the user named root exists on the system. While it is likely true we are not interested in verifying that with this cookbook.

## Update the Test for Port 80

```
~/apache/test/smoke/default/default_test.rb
```

```
# ... FIRST EXAMPLE DELETED ...

# This is an example test, replace it with your own test.
describe port(80), :skip do
  it { should be_listening }
end
```

The other expectation expressed within this file is useful but it is wrong. When we setup a web server we are going to want to have incoming connections on port 80.

So update the following example to state that port 80 should be listening. Also remove the line with the skip as we have now successfully updated the test and I would consider it one that is ours and correct for the code we will eventually write.

## Add a Test to Validate a Working Website

```
~/apache/test/smoke/default/default_test.rb

describe port(80) do
  it { should be_listening }
end

describe command('curl http://localhost') do
  its(:stdout) { should match(/Welcome Home/) }
end
```

+

Ensuring that we are listening on port 80 for incoming connections does not verify that we are in fact returning the correct home page with the welcoming message we plan to write. To do that we will need to write a new expectation.

InSpec provides a helper method that allows you to specify a command. That command returns the results from the command through standard out. We are asking the command's standard out if anywhere in the results match the value 'Welcome Home'.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Examine the cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- Execute the tests and see failure
- Write the recipe to make the test pass
- Execute the tests and see success

With the test defined it is now time to execute the tests and see the failure.

## Move into the Cookbook Directory



```
> cd apache
```

To execute our tests using the tool Test Kitchen we need to be within the directory of the cookbook.

## Review the Existing Kitchen Configuration



```
> cat .kitchen.yml
```

```
---
```

```
driver:
```

```
  name: vagrant
```

```
provisioner:
```

```
  name: chef_zero
```

```
  # You may wish to disable always updating cookbooks in CI or...
```

```
  # For example:
```

```
  #   always_update_cookbooks: <%= !ENV['CI'] %>
```

```
  always_update_cookbooks: true
```

Before we employ Test Kitchen to execute the tests we need make changes to the existing Test Kitchen configuration file. The cookbook was automatically generated with a '.kitchen.yml'.

# The Kitchen Driver

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
verifier:  
  name: inspec  
  
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3
```

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.

Instructor Note: Testing on this remote workstation requires that we use Docker because Vagrant does not work within a virtual environment. Vagrant is the standard choice when working on your local workstation.

# The Kitchen Provisioner

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
verifier:  
  name: inspec  
  
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3
```

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`.

# The Kitchen Verifier

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
verifier:  
  name: inspec  
  
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3
```

This is the framework that is used to verify the state of the system meets the expectations defined.

The third key is the verifier. This verifier by default is using InSpec. Test Kitchen has the ability to use several different verifiers. The default generated with the cookbook generator is InSpec.

# The Kitchen Platforms

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
verifier:  
  name: inspec  
  
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7
```

This is a list of platforms on which we want to apply our recipes.

The fourth 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.

## The Kitchen Suites

```
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3  
  
suites:  
  - name: default  
    run_list:  
      - recipe[apache::default]  
    verifier:  
      inspec_tests:  
        - test/smoke/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 fifth 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'.

# The Kitchen Suites' Run List

```
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3  
  
suites:  
  - name: default  
    run_list:  
      - recipe[apache::default]  
    verifier:  
      inspec_tests:  
        - test/smoke/default  
    attributes:
```

The suite named "default" defines a run\_list.

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

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

## The Kitchen Suites' Tests

```
platforms:  
  - name: ubuntu-16.04  
  - name: centos-7.3  
  
suites:  
  - name: default  
    run_list:  
      - recipe[apache::default]  
    verifier:  
      inspec_tests:  
        - test/smoke/default  
    attributes:
```

This is the path where the InSpec tests can be found.

This is the location where the tests can be found. This is the file that we viewed earlier and updated.

## Remove Settings from the Kitchen Configuration

```
~/apache/.kitchen.yml
```

```
---
```

```
driver:
```

```
  name: vagrant
```

```
provisioner:
```

```
  name: chef_zero
```

```
verifier:
```

```
  name: inspec
```

```
platforms:
```

```
  - name: ubuntu-16.04
```

```
  - name: centos-7
```

The initial Test Kitchen configuration is set up in way for local development on non-virtual machine. Because we are currently on a virtual machine we cannot use vagrant. We are also not interested in those following platforms.

## Add Settings to the Kitchen Configuration

```
~/apache/.kitchen.yml
```

```
---
```

```
driver:
```

```
  name: docker
```

```
provisioner:
```

```
  name: chef_zero
```

```
verifier:
```

```
  name: inspec
```

```
platforms:
```

```
  - name: centos-6.9
```

There are many different drivers that Test Kitchen supports. The docker driver is configured to work on this virtual machine. At this moment we are only interested in verifying that the cookbook we develop works on this current platform.

Later we will return to this configuration file and add an additional platform.

# CONCEPT



## Kitchen List

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.

## View the Test Matrix for Test Kitchen



```
> kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action	Last Checkin
default-centos-69	Docker	ChefZero	InSpec	Ssh	<Not Created>	<Never>

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 test suite and every platform. In our current file we have one suite, named 'default' and one platform CentOS.

Run the following command to verify that the Test Kitchen configuration file had been set up correctly.

# CONCEPT



## Kitchen Create

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

Create one or more instances.

[Create CentOS Instance](#)

©2018 Chef Software Inc. 3-37 

Create or turn on a virtual or cloud instance for the platforms specified in the kitchen configuration.

Running 'kitchen create default-centos-69' would create 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' without any parameters. 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-6.9.

The screenshot shows a slide titled "CONCEPT" with a large orange section containing the title "Kitchen Converge". Below this is a black bar with the command "\$ kitchen converge [INSTANCE|REGEXP|all]". A text box below the command contains the instruction: "Create the instance (if necessary) and then apply the run list to one or more instances." At the bottom, there is a horizontal flowchart with three steps: "Create CentOS Instance" (red), "Install Chef" (orange), and "Apply the Run List" (green). The "Apply the Run List" step is highlighted with a green arrow pointing to it. The slide footer includes the copyright notice "©2018 Chef Software Inc.", page number "3-38", and the Chef logo.

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 apache cookbook's default recipe.

# CONCEPT



## Kitchen Verify

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

Create, converge, and verify one or more instances.

Flowchart:

```
graph LR; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Execute Tests]
```

©2018 Chef Software Inc. 3-39 

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

## Create the Virtual Instance



```
> kitchen create  
----> Starting Kitchen (v1.19.2)  
----> Creating <default-centos-69>...  
      Sending build context to Docker daemon  193 kB  
      Sending build context to Docker daemon  
      Step 0 : FROM centos:centos6  
      centos6: Pulling from centos  
      3690474eb5b4: Pulling fs layer  
      c12ea02d7eb2: Pulling fs layer  
      334af8693ca8: Verifying Checksum  
      334af8693ca8: Download complete  
      273a1eca2d3a: Verifying Checksum
```

Create the instance with the following command. Here Test Kitchen will ask the driver specified in the kitchen configuration file to provision an instance for us.

## Inspect the Virtual Instance



```
> kitchen login
```

```
Last login: Fri Mar 23 15:48:26 2018 from 172.17.42.1
[kitchen@bc530336220c ~]$
```

You can gain access to this virtual instance that we have created through the specified command. The login subcommand allows you to specify a parameter, which is the name of the instance that you want to log into. In your case, you only have one instance so Test Kitchen assumes you want to log into that one.

You are now logged into a virtual instance on a virtual instance.

## Exit the Virtual Instance



```
[kitchen@4eae2dd9e741 ~]$ exit
```

```
logout
```

```
Connection to localhost closed.
```

```
[chef@ip-172-31-14-170 apache]$
```

Logging in to the virtual instance is useful to explore the platform or assist with troubleshooting your recipes they fail in perplexing ways. Right now, we are interested in executing the tests so logout of the instance with the 'exit' command and we will return to the workstation.

## Converge the Virtual Instance



```
> kitchen converge

----> Starting Kitchen (v1.19.2)
----> Converging <default-centos-69>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
----> Installing Chef Omnibus (install only if missing)
    Downloading https://www.chef.io/chef/install.sh to file...
    resolving cookbooks for run list: ["apache::default"]
...
    Finished converging <default-centos-69> (0m27.64s) .
----> Kitchen is finished. (0m28.58s)
```

Creating the instance allows us to view the operating system but Chef is not installed and the cookbook recipe, defined in the run list of the default test suite, has not been applied to the system. To do that you need to run 'kitchen converge'. Converge will take care of all of that.

In this instance the default recipe of the apache cookbook contains no resources. You have not written a single resource that defines your desired state. Before we do that we want to ensure the instance is not already in a state that perhaps already meets the expectations that we defined.

## Execute the Tests Against the Virtual Instance



```
> kitchen verify

-----> Starting Kitchen (v1.19.2)
-----> Setting up <default-centos-69>...
-----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing

Target: ssh://kitchen@localhost:32768

✖ Port 80 should be listening (expected `Port 80.listening?...`)
✖ Command curl localhost stdout should match /Hello, world/...
```

To verify the state of the instance with specification that we defined we use the 'kitchen verify' command. This command will install all the necessary testing tools, configure them, and then execute the test suite, and return to us the results.

Something that is important to mention is that we could have simply run this command from the start. When no previous instance exists, no instance has been created or converged, this command will automatically perform those two steps. When the instance is running, however, the verification step is only run.

# Understanding the Failure Message

```
Target: ssh://kitchen@localhost:32768

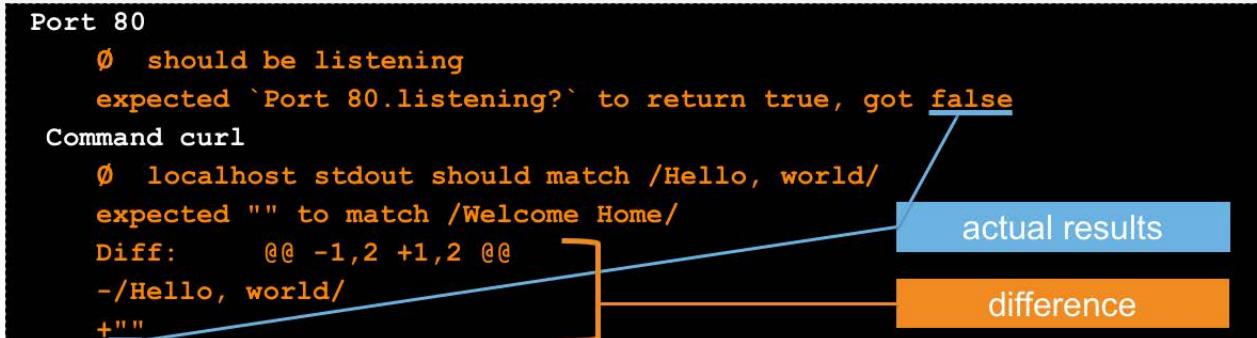
Port 80
  Ø should be listening
    expected `Port 80.listening?` to return true, got false
Command curl
  Ø localhost stdout should match /Welcome Home/
    expected "" to match /Welchome home/
    Diff:      @@ -1,2 +1,2 @@
    -/Hello, world/
    +"""

Test Summary: 0 successful, 2 failures, 0 skipped
>>>> -----Exception-----
>>>> Class: Kitchen::ActionFailed
>>>> Message: 1 actions failed.
>>>>     Verify failed on instance <default-centos-69>. Please see .kitchen/logs/defau...
```

Now, let's read the results from the kitchen verification to ensure that our expectations failed to be met.

## Examine Failure #1

```
Port 80
  Ø should be listening
    expected `Port 80.listening?` to return true, got false
Command curl
  Ø localhost stdout should match /Hello, world/
    expected "" to match /Welcome Home/
    Diff:      @@ -1,2 +1,2 @@
    -/Hello, world/
    +""
```



Each failure is displayed with a human-readable sentence about the defined resource, the expected results, and the result that was received (or 'got').

We see that we have two errors. The first is that port 80 is not listening when we expected to be listening. We also expected the command's standard out to return content to us and it did not; it returned nothing.

## Examine the Test Summary

```
Port 80
  Ø should be listening
    expected `Port 80.listening?' to return true, got false
Command curl
  Ø localhost stdout should match /Welcome Home/
    expected "" to match /Welcome Home/
    Diff:      @@ -1,2 +1,2 @@
    -/Hello, world/
    +"""

Test Summary: 0 successful, 2 failures, 0 skipped
```

A final summary contains the length of execution time with the results shows that RSpec verified 2 examples and found 2 failures.

After all the failures a final summary of the results will be displayed which shows us that our test suite contains 2 examples and that 2 examples failed to meet expectations.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

### Objective:

- ✓ Examine the cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- Write the recipe to make the test pass
- Execute the tests and see success

Now we know for certain that the test instance is not in our desired state. When we write the resources now in the default recipe to bring the instance to the desired state we can be certain that we have done it in a way that meets the expectations that we have established.

# Write the Default Recipe for the Cookbook

```
~/apache/recipes/default.rb

#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright (c) 2018 The Authors, All Rights Reserved.
package 'httpd'

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

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

The following recipe defines three resources. These three resources express the desired state of an apache server that will serve up a simple page that contains the text 'Welcome Home!'.

The package will install all the necessary software on the operating system. The file will create an HTML file with the desired content at a location pre-defined by the web server. The service resource will start the web server and then ensure that if we reboot the system the web server will start up.

## Re-Converge the Virtual Instance



```
> kitchen converge
```

```
-----> Starting Kitchen (v1.19.2)
Converging 3 resources
Recipe: apache::default
  * package[httpd] action install
    - install version 2.2.15-47.el6.centos of package httpd
  * file[/var/www/html/index.html] action create
    - ...
  * service[httpd] action enable
    - enable service service[httpd]
  * service[httpd] action start
    - start service service[httpd]
```

Whenever you make a change to the recipe it is important to run 'kitchen converge'. This command will apply the updated recipe to the state of the virtual instance.

In the output, you should see the resources that you defined being applied to the instance. The package, the file, and the actions of the service.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

### Objective:

- ✓ Examine the cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- ✓ Write the recipe to make the test pass
- Execute the tests and see success

Now with the desired state expressed in the default recipe and applied to the virtual instance it is time to see if the test we wrote initially will now pass. If it does, that means we got everything right in the configuration we wrote in the recipe. We can declare victory!

## Re-Verify the Virtual Instance



```
> kitchen verify
```

```
----> Starting Kitchen (v1.19.2)
----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing
...
Target: ssh://kitchen@localhost:32768

Port 80
  ✓ should be listening
Command curl
  ✓ Command curl localhost stdout should match /Welcome Home/

Summary: 2 successful, 0 failures, 0 skipped
```

To verify the state of the virtual instance you run the 'kitchen verify' command. In the summary you should find the failing expectation no longer fails.

If it does fail, it is time to review the code you wrote in the recipe file and the spec file. When it was failing did you get a different failure than the one that we walked through? That probably means there is an error in the spec file. Did the test instance actually converge successfully? Sometimes output will scroll by and we don't have time to read it. I get it. Scroll back up and see if there was an error message tucked into the 'kitchen converge' you ran.

# EXERCISE



## Build a Reliable Cookbook

*This time it will be different.*

### Objective:

- ✓ Examine the cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- ✓ Write the recipe to make the test pass
- ✓ Execute the tests and see success

So you've done it. You have done Test Driven Development (TDD). Wrote a test. Saw it fail. Wrote a unit of code. Saw it pass.

You created a cookbook. Wrote an expectation in the spec file. Saw the test fail. Wrote a recipe. Applied the recipe. Ran the tests and saw them pass.

# DISCUSSION



## Discussion

What value is there in writing the tests before writing the recipes?

Why is it hard to write the tests before you write the recipe?

Now that you participated in writing a test and then the recipe let's have a discussion.

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?

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

You have performed almost all of the steps of TDD. Next we are going to use the tests to help us refactor the recipe we wrote. In a series of group exercises we will explore some of the important nuances of Test Kitchen's subcommands: converge and verify. And explore another subcommand named: test.



## Refactoring Cookbooks with Tests



©2018 Chef Software Inc.

4-1



We explored the process of developing a test first but to explore the full Test Driven Development (TDD) cycle we need to refactor the code that we wrote.

Refactoring is the process of making changes to the implementation while maintaining the original intention. Without having tests that capture the original intention how do you know if the new implementation did not change the original intention? Fortunately for us we have defined a test that will allow us to make the changes confident that we have not destroyed that original intention.

# CONCEPT



## Test Driven Development

1. Define a test set for the unit first
2. Then implement the unit
3. Finally verify that the implementation of the unit makes the tests succeed.
4. **Refactor**

Refactoring is the often forgotten step in the TDD cycle. When we are able to get our expectations to pass we immediately want to move to our next requirement or next cookbook.

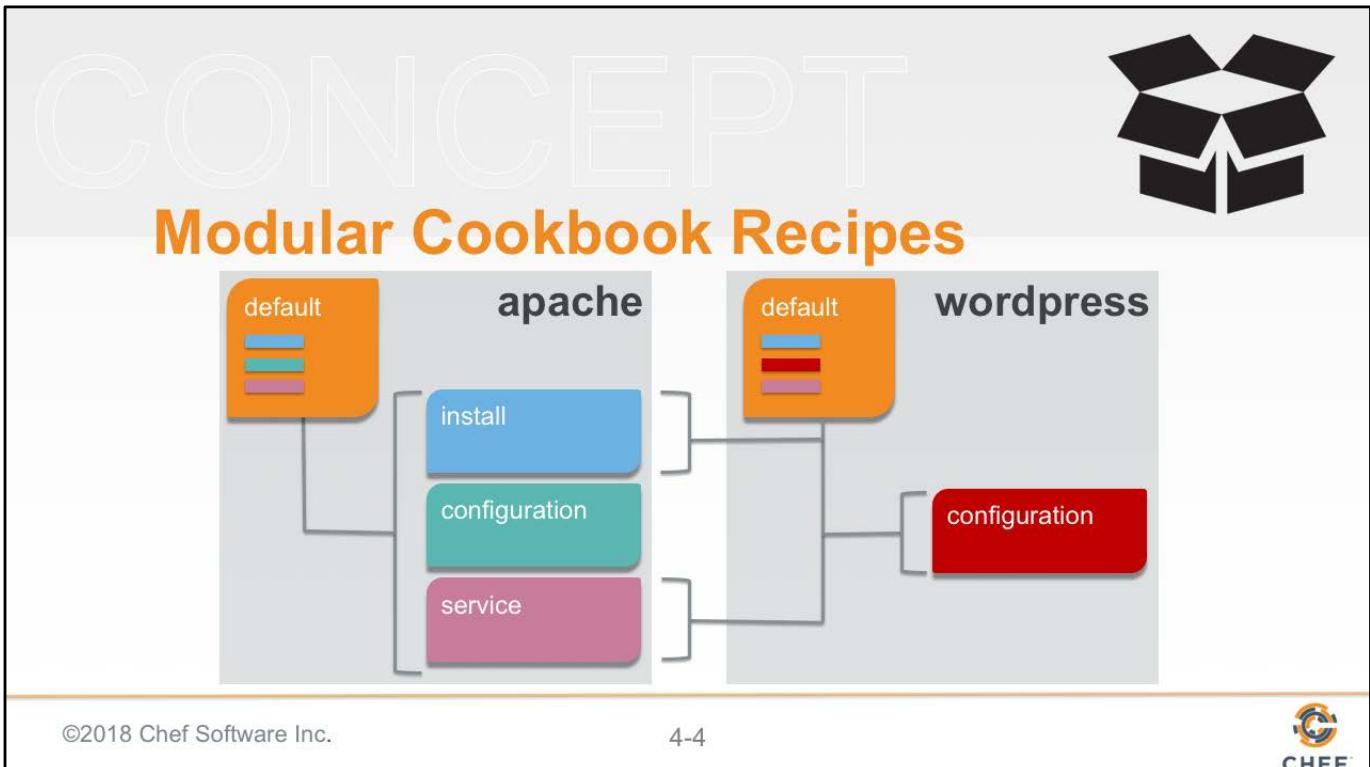
This step is incredibly important. Within it we are able to reflect on the unit of code and tests that we have written and evaluate them. How you evaluate the code may vary based on your experience, the standards defined by the team you work with, or if the code will be shared with the Chef community.

# Objectives

After completing this module, you should be able to:

- Refactor a recipe using `include_recipe`
- Use Test Kitchen to validate the code you refactored
- Explain when to use `kitchen converge`, `kitchen verify` and `kitchen test`.

In this module you will learn how to refactor a cookbook using the method '`include_recipe`', verify the changes with Test Kitchen, and then explain in what scenarios you would choose to use '`kitchen converge`', '`kitchen verify`' and '`kitchen test`'.



Our initial implementation of the default recipe for the apache cookbook defined the entire installation, configuration, and management of the service within a single recipe. This implementation has the benefit of being entirely readable from a single recipe. However, it does not easily allow for other cookbooks that may want to use the apache cookbook to easily choose the components that it may need.

An example of this is that we may deploy wordpress or some other web application that relies on the apache webserver installed and running. In this new cookbook we would like to re-use the resources that installs apache and the resources that manage the service. We most likely do not want to setup a test page that greets people. We are likely going to replace it with application code.

# CONCEPT



## include\_recipe

A recipe can include one (or more) recipes located in cookbooks by using the `include_recipe` method. When a recipe is included, the resources found in that recipe will be inserted (in the same exact order) at the point where the `include_recipe` keyword is located.

<https://docs.chef.io/recipes.html#include-recipes>

The 'include\_recipe' method can be used to include recipes from the same cookbook or external cookbooks. It allows us to accomplish what we saw previously. This gives us the ability to build recipes in more modular ways promoting better re-use patterns within the cookbooks we write.

# CONCEPT

## Recipe Organization

```
recipes/default.rb
include_recipe 'cookbook::install'
include_recipe 'cookbook::configuration'
include_recipe 'cookbook::service'
```

default.rb

install.rb

configuration.rb

service.rb

©2018 Chef Software Inc. 4-6 

To allow better re-use we can choose to refactor a single recipe into more modular recipes that focus on their individual concerns. Then these recipes can be included into the original single recipe through the 'include\_recipe' method.

# EXERCISE

## Refactor to Modular Recipes



*This is why we **can** have nice things!*

**Objective:**

- Refactor the installation into a separate recipe
- Converge the cookbook and execute the tests



You called?

©2018 Chef Software Inc. 4-7

This more modular approach to recipes is very common as the complexity of the cookbook continues to grow. The complexity of the cookbook we are developing is not there, nor will it ever be there for the entirety of this course. However, we are still going to use this opportunity to prematurely optimize to demonstrate the refactoring of a cookbook.

Together we will work through creating a recipe that manages the installation of the webserver.

## Ask Chef About Generating a Recipe



```
> chef generate recipe --help
```

```
Usage: chef generate recipe [path/to/cookbook] NAME [options]
      -C, --copyright COPYRIGHT           Name of the copyright hol...
      -m, --email EMAIL                 Email address of the auth...
      -a, --generator-arg KEY=VALUE     Use to set arbitrary ...
      -I, --license LICENSE            all_rights, apache2, mit, ...
      -g GENERATOR_COOKBOOK_PATH,       Use GENERATOR_COOKBOOK_PA...
          --generator-cookbook
```

First let's return to the chef generator tool and it what information it needs to generate a recipe within a cookbook. The recipe generator can be run from within a cookbook or outside of it. If you are within a cookbook you do not need to specify a path to the cookbook; it's optional.

## Generate an Install Recipe



```
> chef generate recipe install
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/apache/spec/unit/recipes] action create
    (up to date)
  * cookbook_file[/home/chef/apache/spec/spec_helper.rb] action
    create_if_missing (up to date)
  * template[/home/chef/apache/spec/unit/recipes/install_spec.rb]
    action create_if_missing
      - create new file
/home/chef/apache/spec/unit/recipes/install_spec.rb
      - update content in file
/home/chef/apache/spec/unit/recipes/install_spec.rb from none to
187413
```

Since we are within the cookbook directory you simply need to provide it the name of the recipe you want created.

Instructor Note: The generator will create the recipe file with the recipes directory and also a spec file within the unit test directory. Unit testing is a topic that we will discuss in the next module.

## Removing the Generated Test File



```
> rm test/smoke/default/install_test.rb
```

When you use chef, the command-line tool, to generate a recipe it will create three files. First is the recipe file found in the recipes directory. Second is the unit test file found in the 'spec/unit/recipes' directory. Third is the integration test file found in 'test/smoke/default'.

The test file automatically generated for us contains those same examples we saw in the 'default\_test.rb'. We do not want to verify the root user is present and we definitely do not want to verify that port 80 is not listening. So we want to remove this file.

## Write the Install Recipe



~/apache/recipes/install.rb

```
#  
# Cookbook:: apache  
# Recipe:: install  
#  
# Copyright:: 2018, The Authors, All Rights Reserved.  
package 'httpd'
```

The installation of the web server can be expressed with this one resource. Within the new recipe add the following resource.

## Remove the Resource from the Default Recipe

```
~/apache/recipes/default.rb

#
# Cookbook:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
package 'httpd'

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

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

Now that we have defined the installation of the webserver in a separate recipe it is time to remove the installation from the default recipe.

## Include the Install Recipe

```
~/apache/recipes/default.rb

#
# Cookbook:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'

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

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

Replacing it with the 'include\_recipe' method that retrieves the contents of that recipe and includes it here.

# EXERCISE



## Refactor to Modular Recipes

*This is why we **can** have nice things!*

### Objective:

- ✓ Refactor the installation into a separate recipe
- ❑ Converge the cookbook and execute the tests

I see what you did there.



## Re-Converge the Test Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.19.2)
----> Converging <default-centos-69>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
----> Installing Chef Omnibus (install only if missing)
      Downloading https://www.chef.io/chef/install.sh to file...
      resolving cookbooks for run list: ["apache::default"]
      ...
      Finished converging <default-centos-69> (0m27.64s) .
----> Kitchen is finished. (0m28.58s)
```

Whenever a change is made to a recipe or component of the cookbook it is important to converge the latest cookbook against the test instance.

If an error occurs that likely means that you have a typo within your default recipe or the install recipe.

## Re-Verify the Test Instance



```
> kitchen verify
```

```
----> Starting Kitchen (v1.19.2)
----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing
...
Target: ssh://kitchen@localhost:32768

Port 80
  ✓ should be listening
Command curl
  ✓ Command curl localhost stdout should match /Welcome Home/

Summary: 2 successful, 0 failures, 0 skipped
```

If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

# EXERCISE



## Refactor to Modular Recipes

*This is why we **can** have nice things!*

### Objective:

- ✓ Refactor the installation into a separate recipe
- ✓ Converge the cookbook and execute the tests

Nice shave!



# LAB



## The Configuration

- Create a configuration recipe that defines the policy:

```
The file named '/var/www/html/index.html' contains the  
content '<h1>Welcome Home!</h1>'
```

- Delete the automatically generated InSpec test
- Within the default recipe replace the file resource with an include recipe
- Converge and verify the test instance to ensure there are no failures

Now it is your turn to do the same thing for the webserver configuration. The only configuration that we currently perform for the webserver is write out a new default home page. We still want to move that resource to a separate recipe and ensure that we made the change correctly.

When you are done we will review the next few slides together to review your work.

Instructor Note: Another exercise follows this one to manage the service.

Instructor Note: Allow 5 minutes to complete this exercise.

# Generate a Service Recipe



```
> chef generate recipe configuration
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/apache/spec/unit/recipes] action create
    (up to date)
  * cookbook_file[/home/chef/apache/spec/spec_helper.rb] action
    create_if_missing (up to date)
  *
  template[/home/chef/apache/spec/unit/recipes/configuration_spec.rb]
    ] action create_if_missing
      - create new file
/home/chef/apache/spec/unit/recipes/configuration_spec.rb
      - update content in file
/home/chef/apache/spec/unit/recipes/configuration_spec.rb from
```

Generate the configuration recipe within the webserver cookbook.

## Remove the Generated Test File



```
> rm test/smoke/default/configuration_test.rb
```

Remove the automatically generated test file as we are not interested in the sample tests that it generates for us.

# Write the Configuration Recipe

```
~/apache/recipes/configuration.rb

#
# Cookbook:: apache
# Recipe:: configuration
#
# Copyright:: 2018, The Authors, All Rights Reserved.
file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end
```

Define all the resources that are related to the configuration of the webserver within this new recipe

## Remove the Resource from the Default Recipe

```
~/apache/recipes/default.rb

#
# Cookbook:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'

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

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

Remove the resources, that are now defined in the configuration recipe, from the default recipe

# Include the Configuration Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
  
service 'httpd' do  
  action [:enable, :start]  
end
```

Replace the resources that you have removed with an 'include\_recipe' that brings the newly defined configuration recipe.

## Re-Converge the Test Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.19.2)
----> Converging <default-centos-69>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
----> Installing Chef Omnibus (install only if missing)
      Downloading https://www.chef.io/chef/install.sh to file...
      resolving cookbooks for run list: ["apache::default"]
      ...
      Finished converging <default-centos-69> (0m27.64s) .
----> Kitchen is finished. (0m28.58s)
```

The recipe changed so it is important to converge the instance.

## Re-Verify the Test Instance



```
> kitchen verify
```

```
----> Starting Kitchen (v1.19.2)
----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing
...
Target: ssh://kitchen@localhost:32768

Port 80
  ✓ should be listening
Command curl
  ✓ Command curl localhost stdout should match /Welcome Home/

Summary: 2 successful, 0 failures, 0 skipped
```

If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

# LAB



## The Configuration

- ✓ Create a configuration recipe that defines the policy:

```
The file named '/var/www/html/index.html' contains the  
content '<h1>Welcome Home!</h1>'
```

- ✓ Delete the automatically generated InSpec test
- ✓ Within the default recipe replace the file resource with an include recipe
- ✓ Converge and verify the test instance to ensure there are no failures

Congratulations you have successfully refactored the webserver configuration into its own recipe.

# LAB

## The Service



- Create a service recipe that defines the policy:  
  
`The service named 'httpd' is started and enabled`
- Delete the automatically generated InSpec test
- Within the default recipe replace the service resource with an include recipe
- Converge and verify the test instance to ensure there are no failures

One last time!



©2018 Chef Software Inc. 4-27

Now it is your turn to do the same thing for the webserver service.

When you are done we will review the next few slides together to review your work.

Instructor Note: Allow 5 minutes to complete this exercise.

# Generate a Service Recipe



```
> chef generate recipe service
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/apache/spec/unit/recipes] action create
    (up to date)
  * cookbook_file[/home/chef/apache/spec/spec_helper.rb] action
    create_if_missing (up to date)
  * template[/home/chef/apache/spec/unit/recipes/service_spec.rb]
    action create_if_missing
      - create new file
/home/chef/apache/spec/unit/recipes/service_spec.rb
      - update content in file
/home/chef/apache/spec/unit/recipes/service_spec.rb from none to
1f669c
```

Generate the service recipe within the webserver cookbook.

## Remove the Generated Test File



```
> rm test/smoke/default/service_test.rb
```

Remove the automatically generated test file as we are not interested in the sample tests that it generates for us.

## Write the Services Recipe



~/apache/recipes/service.rb

```
#  
# Cookbook:: apache  
# Recipe:: service  
  
#  
# Copyright:: 2018, The Authors, All Rights Reserved.  
service 'httpd' do  
  action [:enable, :start]  
end
```

Define all the resources that are related to the service of the webserver within this new recipe

## Remove the Resource from the Default Recipe

```
~/apache/recipes/default.rb

#
# Cookbook:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
include_recipe 'apache::configuration'

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

Remove the resources, that are now defined in the service recipe, from the default recipe

## Remove the Resource from the Default Recipe

```
~/apache/recipes/default.rb

#
# Cookbook:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
include_recipe 'apache::configuration'
include_recipe 'apache::service'
```

Replace the resources that you have removed with an 'include\_recipe' that brings the newly defined service recipe.

## Re-Converge the Test Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.19.2)
----> Converging <default-centos-69>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
----> Installing Chef Omnibus (install only if missing)
      Downloading https://www.chef.io/chef/install.sh to file...
      resolving cookbooks for run list: ["apache::default"]
      ...
      Finished converging <default-centos-69> (0m27.64s) .
----> Kitchen is finished. (0m28.58s)
```

The recipe changed so it is important to converge the instance.

## Re-Verify the Test Instance



```
> kitchen verify
```

```
----> Starting Kitchen (v1.19.2)
----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing
...
Target: ssh://kitchen@localhost:32768

Port 80
  ✓ should be listening
Command curl
  ✓ Command curl localhost stdout should match /Welcome Home/

Summary: 2 successful, 0 failures, 0 skipped
```

If everything converges successfully it is time to verify the state of the instance with the test that we have defined.



# LAB

## The Service

- ✓ Create a service recipe that defines the policy:

```
The service named 'httpd' is started and enabled.
```

- ✓ Delete the automatically generated InSpec test
- ✓ Within the default recipe replace the service resource with an include recipe
- ✓ Converge and verify the test instance to ensure there are no failures

My hair will grow back!



# DISCUSSION



## Do Our Tests Really Work?

What if we removed code from within the recipes and ran the tests?

During the group exercise and the lab we made changes to the recipes that we were able to verify on the test instance. If you accidentally or purposefully created a typo for yourself you would have seen the converge or the verification fail. However, what if removed code from the recipes that we wrote?

The omission (or in this case removal of code) of resources could have happened. When we refactored the default recipe we may have remembered to remove the resources that manage the configuration but forgot to use the 'include\_recipe' to ensure we loaded the new recipe. Or it is possible that we created a service recipe that we never populated but made all the appropriate changes to the default recipe.

# CONCEPT



## Heckling Your Code

Mutation testing is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways.

Removing code sabotages the policy that you have defined. If you used Test Kitchen to converge and verify the cookbook and saw a failure you can sleep soundly at night knowing your tools have you covered. On the other hand, if Test Kitchen were to return success, after such a change, then it might cause you to break out in a cold sweat.

Removing code from a recipe or recipes is a small change. So is introducing a typo into the code, specifying a different resource name or changing the value of a resource attribute. The process of modifying the code in small ways and then executing the test suite against it is often times referred to as mutation testing.

# EXERCISE



## Heckle That Code

*It could be a game show. Maybe on Twitch?*

**Objective:**

- Remove / Comment source code
- Converge the cookbook and execute the tests

Before we leave this module, let's do a little mutation testing, to ensure the test that we have defined is good enough.

## Comment Out Key Code Within the Default Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
# include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
include_recipe 'apache::service'
```

Return to the default recipe and choose one line to remove or comment out. Here I have chosen to comment out the first line that includes the install recipe.

# EXERCISE



## Heckle That Code

*It could be a game show. Maybe on Twitch?*

**Objective:**

- Remove / Comment source code
- Converge the cookbook and execute the tests

Now with that small mutation in place it is time to converge the cookbook and execute the tests.

## Re-Converge the Test Instance



```
> kitchen converge
```

```
-----> Converging <default-centos-69>...
      Synchronizing Cookbooks:
        - apache (0.1.0)
      Compiling Cookbooks...
      Converging 3 resources
        Recipe: apache::configuration
          (up to date)
        Recipe: apache::service
          (up to date)
          * service[apache] action enable (up to date)
```

When converging the updated recipe it no longer shows the install recipe being loaded. This has changed the number of resources that are converged on the test instance. Removing the recipe from the default recipe does not remove any of the components that it previously installed.

## Re-Verify the Test Instance



```
> kitchen verify
```

```
----> Starting Kitchen (v1.19.2)
----> Verifying <default-centos-69>...
      Use `/home/chef/apache/test/smoke/default` for testing
...
Target: ssh://kitchen@localhost:32768

Port 80
  ✓ should be listening
Command curl
  ✓ Command curl localhost stdout should match /Welcome Home/

Summary: 2 successful, 0 failures, 0 skipped
```

Verification of the test instance will return a success. Despite removing the install recipe from the default recipe the test instance is still able to serving the default web page that our test is looking for when it requests data from the site.

# CONCEPT



## Kitchen Converge & Verify

Running converge or verify will create a new instance the first time it is run.  
The same instance is used for each additional converge or verify.

The test instance policy changed, but no resource explicitly removed or  
uninstalled the resources defined in the install recipe.

This is important feature and limitation of using Test Kitchen's 'converge' and 'verify'. Both of these commands will create a test instance the first time they are executed. Every time after these commands will use the same test instance again and again.

When we remove resources from a recipe we do not explicitly uninstall them from the test instance. We simply do not enforce their policy anymore. On an existing system, which this test instance is after the first run, this means it is actually in the desired state that we no longer define. That means that the webserver is still installed, the default web page has still been updated, and the service is still running.

To ensure our cookbook works on a new system it is important to delete the test instance and start over.

# CONCEPT



## Kitchen Destroy

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

Destroys one or more instances.



©2018 Chef Software Inc. 4-44 

Test Kitchen provides the 'destroy' subcommand. Destroy is available at all stages and essentially cleans up the instance. This is useful when you make changes to the configuration policy you define and you want to ensure that it will work on a brand new instance.

Instructor Note: It works as all the other commands do with regard to parameters and targeting instances.

# CONCEPT

## Kitchen Test

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

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

©2018 Chef Software Inc. 4-45 

Test Kitchen also provides the subcommand 'test'. Test provides one command that wraps up all the stages in one command. It will destroy any test instance that exists at the start, create a new one, converge the run list on that instance, and verify it. If everything passes the 'test' subcommand will finish by destroying that instance. If it fails at one of these steps it usually leaves the instance running to allow you to troubleshoot it.

Instructor Note: It works as all the other commands do with regard to parameters and targeting instances.

# CONCEPT



## Kitchen Test

Destroying the instance ensures that the policy is being applied to a new instance.

The test instance is re-created and then the updated policy is applied to the new instance. The new policy is incomplete causing an error.

Running 'kitchen test' is useful if want to ensure the policy you defined works on a new instance.

## Test the Cookbook Against a New Instance



```
> kitchen test
```

```
----> Starting Kitchen (v1.19.2)
----> Cleaning up any prior instances of <default-centos-69>
----> Destroying <default-centos-69>...
...
[TIMESTAMP] FATAL: Chef::Exceptions::ChildConvergeError:
Chef run process exited unsuccessfully (exit code 1)
>>>>> -----Exception-----
>>>>> Class: Kitchen::ActionFailed
>>>>> Message: 1 actions failed.
>>>>>     Converge failed on instance <default-centos-69>.
```

Running 'kitchen test' in this instance will expose the issue that we created by removing that installation of the webserver. This is because the new instance no longer installed the necessary packages so the file path was never created for the default HTML file and there are no services to run.

The test that you wrote correctly verifies the state of the system. What is important to notice is that there are important differences in the Test Kitchen commands.

## Converge & Verify

**Faster** execution time

Running converge twice will ensure your policy applies without error to **existing instances**

## Test

**Slower** execution time

Running test will ensure your policy applies without error to any **new instances**

Using Test Kitchen to run 'kitchen converge' and 'kitchen verify' is much faster because you are essentially applying and verifying the policy that you have defined against an already running instance. The drawback is that only running 'converge' and 'verify' will not demonstrate for you how your policy will act on a brand new instance.

Using Test Kitchen to run 'kitchen test' is slower because every time you are recreating the test instance, installing chef, and applying the policy on that new instance. The drawback here is the longer feedback cycle and only running 'test' will not demonstrate for you how your policy will act on an existing instance.

# EXERCISE



## Heckle That Code

*It could be a game show. Maybe on Twitch?*

**Objective:**

- ✓ Remove / Comment source code
- ✓ Converge the cookbook and execute the tests

Removing code and causing a failure showed us some of the differences between 'kitchen converge and verify' and 'kitchen test'. To ensure that we understand these important differences let's have a discussion.

# DISCUSSION



## Discussion

What is happening when running `kitchen test`?

What types of bugs would `kitchen converge` & `kitchen verify` find when running?

What is the difference between `kitchen test` and running both `kitchen converge` & `kitchen verify` together?

How long do each of these approaches take?

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?

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

You have performed the complete TDD cycle from start to finish. Now that you have seen this cycle and understand that we are simply going to continue to repeat it as we develop cookbooks it is important to talk about the amount of time it takes for us to get feedback. In the next section we are going to explore that further by introducing a new testing tool and language that promises to give us faster feedback.



## Faster Feedback with Unit Testing

©2018 Chef Software Inc.

5-1



If you are planning on adopting Test Driven Development and use it to validate most if not all of the changes that you make to a cookbook you now have to are welcoming into your workflow the interruption of running the tests. Testing provides value as it validates the work that you accomplish but it is still an interruption.

# PROBLEM



## Slower Feedback Cycle

The slower the feedback loop the less value it provides to you while developing your cookbooks. You are less inclined to run the test suite. Which means you will likely miss issues as they happen.

Interruptions are not conducive to helping you building a flow. To help reduce the interruptive nature of testing we can look at ways to decrease the amount of time you have to wait to receive the feedback from the tests. A faster feedback cycle will increase your likelihood of seeking that feedback again for smaller sets of changes. Slower feedback cycles will increase your likelihood of seeking feedback less often. Causing you create larger sets of changes which has the chance of masking potential issues.

# Objectives

After completing this module, you should be able to:

- Explain the importance and limitations of unit testing
- Write and execute a unit test

In this module you will learn the importance and limitations of unit testing as you write and execute unit tests to help increase the rate at which you receive feedback.

# CONCEPT

## External Dependencies

The speed of the test suite is affected by the external dependency on the creation of the test instance, installing chef, and applying the run list.

Create CentOS Instance

Install Chef

Apply the Run List

Build Node (ohai)

Synchronize Cookbooks

Build Resource Collection

Converge

Execute Tests

The reason that the feedback cycle takes as long as it does with Test Kitchen is because of the external requirements. Creating the test instance, installing chef, and then applying the run list provide real value because we are able to see the recipe being applied to a virtual instance. However, all these external dependencies incur a time cost as we wait for the network to download images or packages, the test instance's processor to calculate keys or data, or the file-system to create files and folders.

# CONCEPT

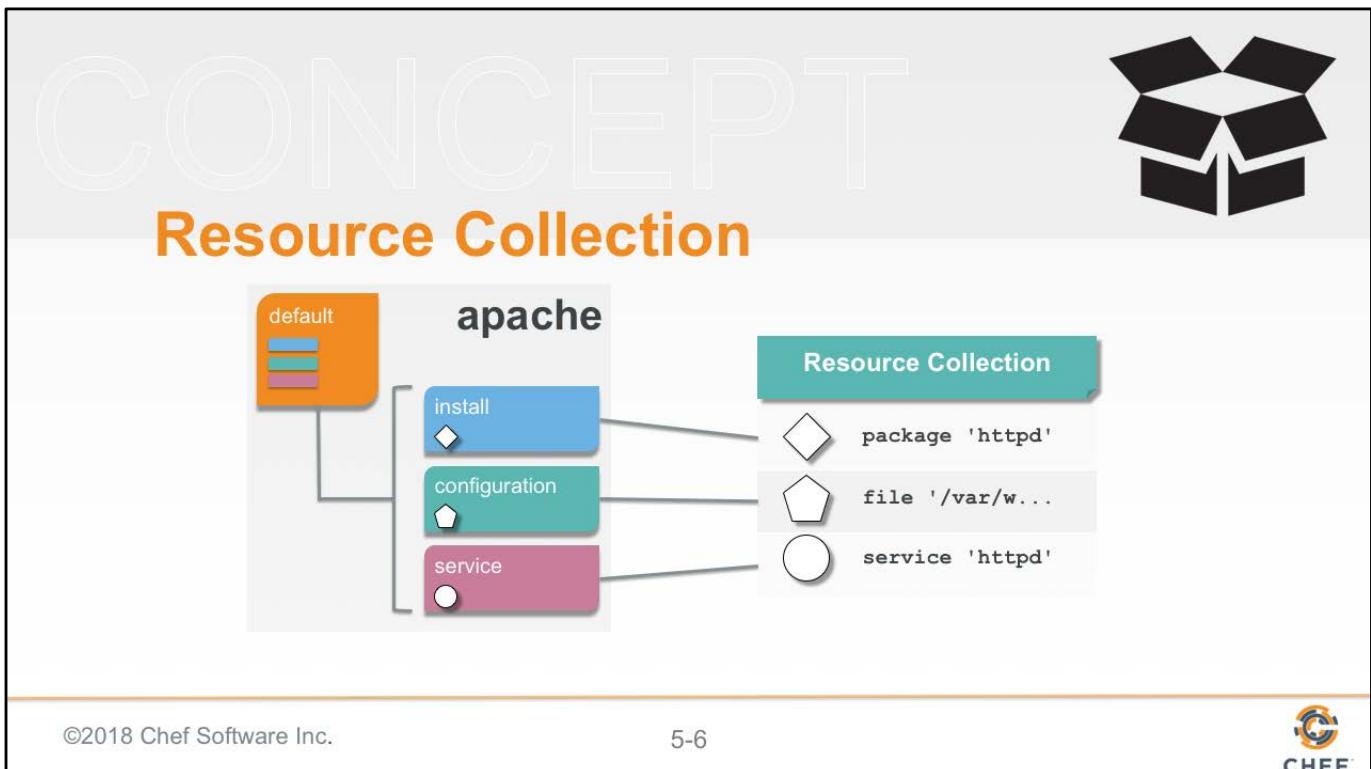
## Build Resource Collection

The resource collection is a list of all the resources and recipes loaded across all the recipes within the run list.

```
graph TD; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Build Node (ohai)]; D --> E[Synchronize Cookbooks]; E --> F[Build Resource Collection]; F --> G[Converge]; G --> H[Execute Tests]
```

©2018 Chef Software Inc. 5-5 

When we mutated our code and executed the test suite we created issues with the resources that we defined and recipes that we included. These changes affected the resources that were applied to the system by omitting resources from the 'Resource Collection'. If we were able to remove the external dependencies and focus on the state of the Resource Collection we would be able to determine if there were problems with the recipes we wrote without the need of any of those external dependencies.



But first let's talk more about the 'Resource Collection' ...

After a cookbook and its recipes have been synchronized the majority of the cookbook content is loaded into memory by 'chef-client'. The recipes defined on the run list are evaluated during this time and the resources found within the recipes and any included recipes, are added to a resource collection. They are not immediately executed like one might assume.

The 'Resource Collection' is almost like a to-do list for the node. It contains the list of all the resources, in order, that need to be accomplished to bring the instance into the desired state. Later, in the converge step, the resources defined in the Resource Collection are executed and perform their various forms of test-and-repair to bring the instance into the desired state.

# CONCEPT



## RSpec and ChefSpec

RSpec is a Domain Specific Language (DSL) that allows you to express and execute expectations. These expectations are expressed in examples that are asserted in different example groups.

ChefSpec provides helpers and tools that allow you to express expectations about the state of **resource collection**.

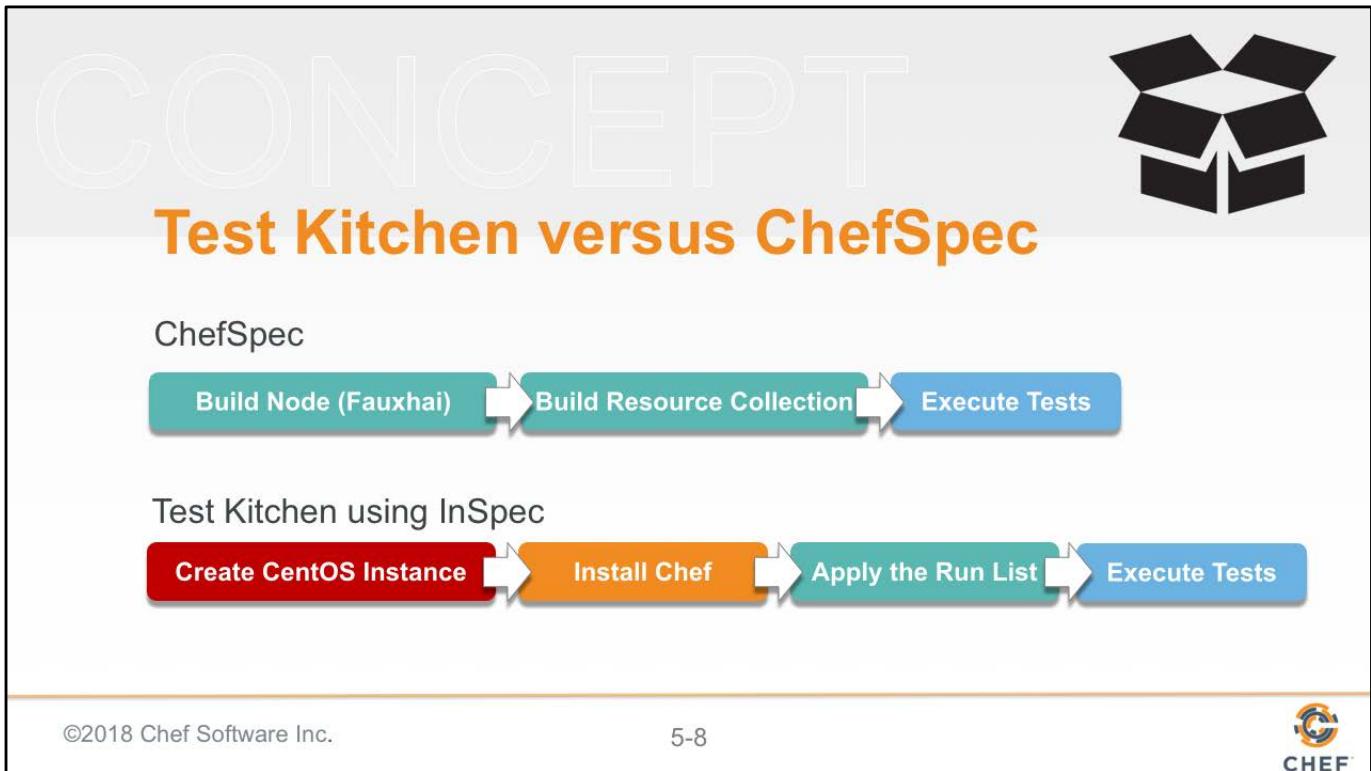
ChefSpec

RSpec

Chef

Ruby

ChefSpec provides a method for us to create an in-memory execution of applying the run list, building the resource collection, and then setting up expectations about the state of the resource collection. ChefSpec, similar to InSpec is built on top of RSpec; relying on it to provide the core framework and language. The benefit to us is that a lot of the same language constructs are employed.



Verifying the resource collection with ChefSpec requires far fewer external dependencies and that allows us to get feedback faster but at the cost of not applying the recipes we write against a test instance. This opens us up to situations where we could compose recipes and execute examples that are shown to work because they were correctly added to the resource collection but fail when it comes time for the recipes to apply the desired state against a test instance.

# EXERCISE



## Faster Feedback While Developing Cookbooks

*The faster the feedback from our tests, the more likely we are to run them.  
The more likely we are to run them means they will catch more issues.*

### Objective:

- Review and run the existing tests
- Identify the tests that we need to write
- Write and execute the tests to identify the failure
- Fix the code and execute the tests to see success

We have the integration test, the one defined in InSpec, executed through Test Kitchen to ensure the recipes we write behave as we expect on the test instances we define. The benefit of writing tests focused around the Resource Collection will allow us to gain feedback quickly and build a better development workflow.

This next group exercise we will review the existing ChefSpec specifications defined for us and how we can expand them to capture our additional expectations about the Resource Collection.

## View the Spec Directory

 > tree spec

```
spec
├── spec_helper.rb
└── unit
    └── recipes
        ├── configuration_spec.rb
        ├── default_spec.rb
        ├── install_spec.rb
        └── service_spec.rb
2 directories, 5 files
```

When generating recipe files we were also given a matching specification file in the 'spec/unit' directory. The ChefSpec defined specifications are all contained within this directory.

## View the Test for the Default Recipe

```
~/apache/spec/unit/recipes/default_spec.rb

require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on an Ubuntu 16.04' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect(chef_run).to_not raise_error
    end
  end
end
```

Open up the default specification file and lets read through and begin to understand the initial expectation that is automatically defined.

The expectations defined in this initially generated specification file should look a little familiar. This is because ChefSpec is built on Rspec. Similar to how InSpec is built. ChefSpec requires a little more setup as we are creating an in-memory execution.

## View the Test for the Default Recipe

```
~/apache/spec/unit/recipes/default_spec.rb
```

```
require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on an Ubuntu 16.04' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect(chef_run).to_not raise_error
    end
  end
end
```

It is often common for specification files to share similar functionality. As your suite of examples grows you will often move common, shared expectations and helpers to a common file that is required here at the top of the file. This will load the contents of the 'spec\_helper' file found within the root of the 'spec' directory.

ChefSpec employs RSpec's example groups to describe the cookbook's recipe. This is stating that the examples we defined within this outer example group all relate to the apache cookbook's default recipe. Within this example group we see another context that is defined. This time using the method 'context'. 'context' and 'describe' are exactly same in almost every way. A lot of developers like to use context as it more clearly states that the example group is focused on a particular scenario. In this instance the particular scenario we are going to be specifying examples in a scenario where all the attributes are default on CentOS 6.9.

Instructor Note: 'describe' and 'context' are almost completely interchangeable with one exception. 'context' cannot be used as the outermost example group.

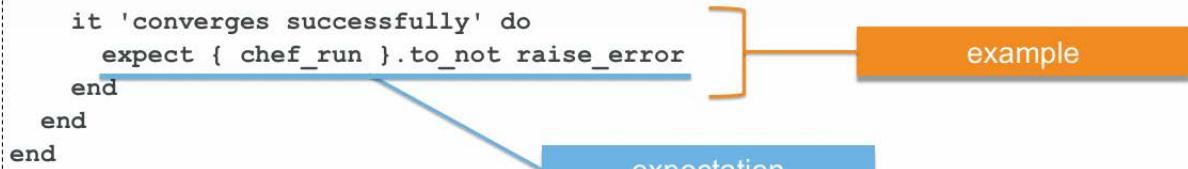
## View the Test for the Default Recipe

```
~/apache/spec/unit/recipes/default_spec.rb
```

```
require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on an Ubuntu 16.04' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

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



expectation

example

Within the inner context we finally set the stage for us to define our examples with their expectations. There is a single example defined and that is stating that when the chef run evaluates and creates the resource collection it should do so without raising an error. A situation that might raise an error is if we included a recipe that does not exist or if we were to use a resources type that does not exist.

## View the Test for the Default Recipe

```
~/apache/spec/unit/recipes/default_spec.rb
```

```
require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on an Ubuntu 16.04' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described recipe)
    end

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

The code shows a ChefSpec test for the 'apache::default' recipe. It uses the 'describe' block to define the recipe and its context. Inside, a 'let(:chef\_run)' block defines an instance of 'ChefSpec::ServerRunner' with 'platform: ubuntu' and 'version: 16.04'. The 'runner.converge(described recipe)' line runs the converge operation on the runner object, passing the 'described recipe' as a parameter. The 'it' block then asserts that the run does not raise an error.

The 'chef\_run' helper there is being provided by the 'let' defined above the example within the same context. Defining the 'chef\_run' in the 'let' above is done with a Ruby Symbol. This is simply naming it so that it can be used within any of the examples in the current context and even sub-contexts. The helper is simply executing some code that sets up an in-memory chef-client run with a Chef Server.

The 'ServerRunner' is a class defined within the 'ChefSpec' namespace. All Ruby classes have the method 'new' which will return an object which is a new instance of that described class. The object is stored in a local variable, named 'runner', which immediately invokes a method 'converge' with a single parameter 'described\_recipe'

The parameter 'described\_recipe' refers to the recipe defined in the outermost describe. This is mostly for convenience so that we do not have to redefine the same String multiple times within the same specification file.

The goal of this single, boilerplate example is very simple: perform a chef-client run and ensure there are no errors. Now, let's execute this specification.

## Execute the Test for the Default Recipe



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

```
.
```

```
Finished in 0.8208 seconds (files took 2.5 seconds to load)
```

```
1 example, 0 failures
```

passing example

To execute the specification file defined you will need to run the command 'rspec'. The 'rspec' command was installed with the Chef Development Kit (ChefDK) on the workstation. It is contained in an additional folder of tools embedded within the ChefDK that are not added to the system path. This is because some Chef developers are Ruby developers and may already have a version of RSpec installed. Specifying the 'chef exec' as a prefix loads the context of all these embedded tools and allows them to be executed on the command-line.

The 'rspec' command accepts many parameters. The most important one is used here and that is specifying the file path to the specification we want to execute. When the command executes a summary of the executed examples will be displayed at the bottom. At this moment it looks like the one expectation completes successfully. The chef run completes without any errors.

Instructor Note: On the workstations the learners do not need to prepend the rspec command with 'chef exec'. This is because 'rspec' and all the other tools embedded in the ChefDK have been added to the path. On a learner's local system this is likely not the case and so they will need to type this entire command with prefix.

# EXERCISE



## Faster Feedback While Developing Cookbooks

*The faster the feedback from our tests, the more likely we are to run them.  
The more likely we are to run them means they will catch more issues.*

### Objective:

- ✓ Review and run the existing tests
- Identify the tests that we need to write
- Write and execute the tests to identify the failure
- Fix the code and execute the tests to see success

We have the language and the tool that will allow us to express our expectations. We now need to examine the recipe again to see what example or examples we want to define within the specification file.

## These are the Three Things to Test

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook Name:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
# include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
include_recipe 'apache::service'
```

Within the default recipe we commented out the line that included the install recipe from the apache cookbook. This seems like an expectation that we want to define. When converging the default recipe we expect that the install recipe from the apache cookbook would be included.

We do not yet know how to define this expectation but we know the work that we want to accomplish. So lets take this one step at a time then and first capture the description for the example even if we do not yet know how to express the expectation.

# Update the ChefSpec Platform

```
~/apache/spec/unit/recipes/default_spec.rb

require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on CentOS 6.9' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')+
      runner.converge(described_recipe)
    end

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

Recall that unit tests are a check of the resource collection. So it actually does not matter what platform you specify here. Unit tests are only using those platform values to give you a generic set of Ohai data for that platform. If it's no harm, then why make this change?

Because these tests are your documentation. To leave it as the default value would let your tests pass but they would give a conflicting message about where this cookbook can run. Ensuring the right platform becomes important when your recipes require switching on platforms.

## Create a Pending Test

```
~/apache/spec/unit/recipes/default_spec.rb

# ... START OF THE SPEC FILE ...
it 'converges successfully' do
  expect { chef_run }.to_not raise_error
end

it 'includes the install recipe'

end
end
```

Returning to the specification we can describe the example that we want to create without having to know how to define the expectation by defining an example without the block. RSpec treats these examples without the associated block as a pending test.

This is an incredibly useful feature when you want to start expressing your examples. This allows you to quickly identify all the examples without getting mired in the details of their implementation.

## Execute the Tests to See the Pending Tests



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

```
.*
```

```
pending example
```

```
Pending: (Failures listed here are expected and do not affect your suite's status)
```

```
1) apache::default When all attributes are default, on CentOS 6.9 includes the install recipe
```

```
# Not yet implemented
```

```
# ./spec/unit/recipes/default_spec.rb:22
```

```
# ... OUTPUT CONTINUES ON NEXT SLIDE ...
```

When executing 'rspec' again we should see the new pending example that we defined within the specification file.

## Execute the Tests to See the Pending Tests



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

```
.*
```

Pending: (Failures listed here are expected and do not affect your suite's status)

summary

1) apache::default When all attributes are default, on CentOS 6.9 includes the install recipe

# Not yet implemented

# ./spec/unit/recipes/default\_spec.rb:22

# ... OUTPUT CONTINUES ON NEXT SLIDE ...

spec file : line number

RSpec's pending summary is similar to the failure summary. The pending examples are identified and then finally they are collected together in list. Each pending example will show the words you used in the description text in a single sentence. Below that it will state the example is not yet implemented and then finally display the file path and line number of where it can be found.

## View the Results to See the Pending Tests



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

```
# ... OUTPUT CONTINUED FROM PREVIOUS SLIDE ...
```

```
Finished in 0.97525 seconds (files took 1.81 seconds to load)
```

```
2 examples, 0 failures, 1 pending
```

The summary will now display that an additional example has been added and it will be reported as being set to pending.

# EXERCISE



## Faster Feedback While Developing Cookbooks

*The faster the feedback from our tests, the more likely we are to run them.  
The more likely we are to run them means they will catch more issues.*

### Objective:

- ✓ Review and run the existing tests
- ✓ Identify the tests that we need to write
- Write and execute the tests to identify the failure
- Fix the code and execute the tests to see success

Now that we have defined the pending example, setting up the work for ourselves, it is time to learn how to express the expectation.

# REFERENCE



## ChefSpec Documentation

Find within the documentation examples of testing for `include_recipe`.

- Search the README
- Search through the 'examples' directory

<https://github.com/chefspec/chefspec>

To understand how to express an expectation we need to go to the documentation. The ChefSpec README provides a wealth of examples in the README. In the past an 'include\_recipe' example has been one of the many examples shared in the README. Use the search feature of your browser to find it within the document.

If it is not there, the ChefSpec project has a top-level folder named 'examples' which contains examples for nearly every feature that ChefSpec is able to define expectations. Searching through there you will find a folder titled 'include\_recipe', within it should a folder the shows the recipes and the matching specifications.

## Write the Test that Verifies the Include Recipe

```
~/apache/spec/unit/recipes/default_spec.rb

# ... START OF THE SPEC FILE ...
it 'converges successfully' do
  expect { chef_run }.to_not raise_error
end

it 'includes the install recipe' do +  
  expect(chef_run).to include_recipe('apache::install')
end

end
end
```

Returning to the specification file we now need to expand the example to include the expectation we want to write. To do that we add a 'do' to the end of the example. We move to the next line, indent two spaces and then define the following expectation. The expectation uses a natural language way of expressing the expectation. Here we are expressing the expectation that the 'chef\_run' includes the recipe with the specified name.

## Execute the Tests to See the Failure



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

```
. F
```

Failures:

failing example

```
1) apache::default When all attributes are default, on CentOS  
6.9 includes the install recipe  
Failure/Error: expect(chef_run).to  
include_recipe('apache::install')  
expected ["apache::default", "apache::configuration",  
"apache::service"] to include "apache::install"  
# ./spec/unit/recipes/default_spec.rb:21:in `block (3 levels)  
in <top (required)>'
```

With the example defined with the expectation when we execute 'rspec' we see the failure that eluded us we ran 'kitchen converge & verify' on an existing very quickly.

The failure summary here is similar to the failure summary return by RSpec when employed by Test Kitchen. The example is displayed, the expectation is expressed, the failure to meet expectation and file name and line number within the file where to find the expectation.

# EXERCISE



## Faster Feedback While Developing Cookbooks

*The faster the feedback from our tests, the more likely we are to run them.  
The more likely we are to run them means they will catch more issues.*

### Objective:

- ✓ Review and run the existing tests
- ✓ Identify the tests that we need to write
- ✓ Write and execute the tests to identify the failure
- ❑ Fix the code and execute the tests to see success

Now that we have a failing test it is time to fix the problem.

## Uncomment the Include Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook Name:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
include_recipe 'apache::service'
```

Returning to the default recipe it is time to restore the code that we previously commented out.

## Execute the Tests to See it Pass



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

```
.
```

```
Finished in 0.97525 seconds (files took 1.81 seconds to load)
2 examples, 0 failures
```

Executing 'rspec' one more time should show the previous failing example now as a passing example.

# EXERCISE



## Faster Feedback While Developing Cookbooks

*The faster the feedback from our tests, the more likely we are to run them.  
The more likely we are to run them means they will catch more issues.*

### Objective:

- ✓ Review and run the existing tests
- ✓ Identify the tests that we need to write
- ✓ Write and execute the tests to identify the failure
- ✓ Fix the code and execute the tests to see success

Now we can confidently state that the default recipe includes the install recipe and we can receive this verification in a faster feedback cycle than we saw with running 'kitchen test'.

Mutation testing is not Test Driven Development (TDD) but the act that we performed was fairly close. This is a tactic that is useful when you are writing expectations for already defined recipes for existing cookbooks or when it feels near impossible to start with the tests first. This process does one of the important aspects of TDD which is ensure the expectations we set correctly capture the state of the code.

# LAB



## Continue with Mutation Testing

- Comment out the next line in the apache cookbook's default recipe
- Write the example with expectation that will generate a failure
- Verify that one example generates a failure
- Restore the code in the recipe
- Verify that all examples pass

❖ Repeat this series of steps for each line within the default recipe

There are few more chances to reinforce this process. As an exercise continue mutating the code within the default recipe, defining the expectations, and then fixing the code. Create a single mutation at a time and become focus on understanding the process of moving between files and executing commands.

Instructor Note: Allow 10 minutes to complete this exercise

Instructor Note: The learners could accomplish both of tasks at the same time. They likely will want to do that. I would encourage you have them perform the steps separately as it will emphasize the activity of moving between the recipe, the specification file, and their shell.

## Comment the Include Recipe

```
~/apache/recipes/default.rb

#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
# include_recipe 'apache::configuration'
include_recipe 'apache::service'
```

Let's review by walking through one more example within the default recipe. Another line within the recipe is similar to the first one except it is concerned with the inclusion of the configuration recipe. Here it is commented out.

## Write the Test that Verifies the Include Recipe

```
~/apache/spec/unit/recipes/default_spec.rb

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

it 'includes the install recipe' do
  expect(chef_run).to include_recipe('apache::install')
end

it 'includes the configuration recipe' do
  expect(chef_run).to include_recipe('apache::configuration')
end

end
end
```

Returning to the specification file to define the example and the new expectation.

## Execute the Tests to See it Fail



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

..F

Failures:

  1) apache::default When all attributes are default, on CentOS
  6.9 includes the configuration recipe
      Failure/Error: expect(chef_run).to
      include_recipe('apache::configuration')
          expected ["apache::default", "apache::configuration",
          "apache::install", "apache::service"] to include
          "apache::configuration"
      # ./spec/unit/recipes/default_spec.rb:27:in `block (3 levels)
      in
```

Seeing the failure when executing the 'rspec' command.

## Uncomment the Include Recipe

```
~/apache/recipes/default.rb

#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
include_recipe 'apache::configuration' #include_recipe 'apache::service'
```

Restoring the code to its previous state

## Execute the Tests to See it Pass



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

```
...
```

```
Finished in 0.97525 seconds (files took 1.81 seconds to load)
3 examples, 0 failures
```

Executing 'rspec' again to verify that the expectations have been met successfully

## Comment the Include Recipe

```
~/apache/recipes/default.rb

#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
include_recipe 'apache::configuration'
include_recipe 'apache::service'
```

Restoring the code to its previous state

## Write the Test that Verifies the Include Recipe

```
~/apache/spec/unit/recipes/default_spec.rb

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

it 'includes the install recipe' do
  expect(chef_run).to include_recipe('apache::install')
end

it 'includes the configuration recipe' do
  expect(chef_run).to include_recipe('apache::configuration')
end

it 'includes the service recipe' do
  expect(chef_run).to include_recipe('apache::service')
end
end
end
```

Returning to the specification file to define the example and the new expectation.

## Execute the Tests to See it Fail



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

```
...F
```

```
Failures:
```

```
1) apache::default When all attributes are default, on CentOS  
6.9 includes the service recipe
```

```
Failure/Error: expect(chef_run).to  
include_recipe('apache::service')  
expected ["apache::default", "apache::install",  
"apache::configuration"] to include "apache::service"  
# ./spec/unit/recipes/default_spec.rb:31:in `block (3 levels)
```

Seeing the failure when executing te 'rspec' command.

## Uncomment the Include Recipe

```
~/apache/recipes/default.rb

#
# Cookbook Name:: apache
# Recipe:: default
#
# Copyright:: 2018, The Authors, All Rights Reserved.
include_recipe 'apache::install'
include_recipe 'apache::configuration'
include_recipe 'apache::service'
```

Restoring the code to its previous state

## Execute the Tests to See it Pass



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

```
....
```

```
Finished in 0.97525 seconds (files took 1.81 seconds to load)
4 examples, 0 failures
```

Executing 'rspec' again to verify that the expectations have been met successfully

# LAB



## Continue with Mutation Testing

- ✓ Comment out the next line in the apache cookbook's default recipe
- ✓ Write the example with expectation that will generate a failure
- ✓ Verify that one example generates a failure
- ✓ Restore the code in the recipe
- ✓ Verify that all examples pass

❖ Repeat this series of steps for each line within the default recipe

There are more mutations that you could try within the default recipe and other recipe files that exist within the cookbook but this is a good point to stop and enjoy the work that you have accomplished.

The feedback cycle on using Rspec to execute ChefSpec examples returns results faster than we saw with Test Kitchen and gives us a good understanding of what is being added to the 'Resource Collection'.

Let's have a discussion.

# DISCUSSION



## Discussion

What functionality did you test in the integration tests?

What functionality did you test in these unit tests?

What do you see as the scope of unit testing versus integration testing?

What are the differences between a ChefSpec test and a InSpec test?

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?

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

We have the faster feedback that we set out to create for us at the beginning of this section. We were able to verify the work being performed in the default recipe. Now it is time to focus our attention on the remaining recipes within the cookbook and set up expectations on the resources that they define.



## Testing Resources in Recipes

©2018 Chef Software Inc.

6-1



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 0.97525 seconds (files took 1.81 seconds to load)
```

```
2 examples, 0 failures
```

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

## Delete context for Ubuntu

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

require 'spec_helper'

describe 'apache::install' do
  context 'When all attributes are default, on Ubuntu 16.04' do
    let(:chef_run) do
      # for a complete list of available platforms and versions see:
      # https://github.com/customink/fauxhai/blob/master/PLATFORMS.md
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      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.

## 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 CentOS 6.9' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')+
      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 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/chefspec/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 0.62738 seconds (files took 1.84 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:: apache  
# Recipe:: install  
#  
# Copyright:: 2018, 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 CentOS  
6.9 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:: apache
# Recipe:: install
#
# Copyright:: 2018, 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

## Delete context for Ubuntu

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

require 'spec_helper'

describe 'apache::service' do
  context 'When all attributes are default, on Ubuntu 16.04' do
    let(:chef_run) do
      # for a complete list of available platforms and versions see:
      # https://github.com/customink/fauxhai/blob/master/PLATFORMS.md
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

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

# Update the ChefSpec Platform

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

require 'spec_helper'

describe 'apache::service' do
  context 'When all attributes are default, on CentOS 6.9' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')+
      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 0.6329 seconds (files took 1.85 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.

## Delete context for Ubuntu

```
~/apache/spec/unit/recipes/configuration_spec.rb

require 'spec_helper'

describe 'apache::configuration' do
  context 'When all attributes are default, on Ubuntu 16.04' do
    let(:chef_run) do
      # for a complete list of available platforms and versions see:
      # https://github.com/customink/fauxhai/blob/master/PLATFORMS.md
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

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

# Update the ChefSpec Platform

```
~/apache/spec/unit/recipes/configuration_spec.rb

require 'spec_helper'

describe 'apache::configuration' do
  context 'When all attributes are default, on CentOS 6.9' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')+
      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 Configuration

```
~/apache/spec/unit/recipes/configuration_spec.rb

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

it 'creates the index.html' do
  expect(chef_run).to render_file('/var/www/html/index.html').with_content('<h1>Welcome Home!</h1>')
end
end
end
```

Let's review the final resulting specification for only the configuration recipe. We defined an example that states we expect the chef run to render a file (either through a file, template, or cookbook\_file) with the correct content.

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

# 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 be found by RSpec.

## Execute All the Tests in the Spec Directory



```
> chef exec rspec
```

```
.....
```

```
Finished in 2.27 seconds (files took 1.82 seconds to load)
```

```
11 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 6.9 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?

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.



## Testing While Refactoring to Attributes



©2018 Chef Software Inc.

7-1



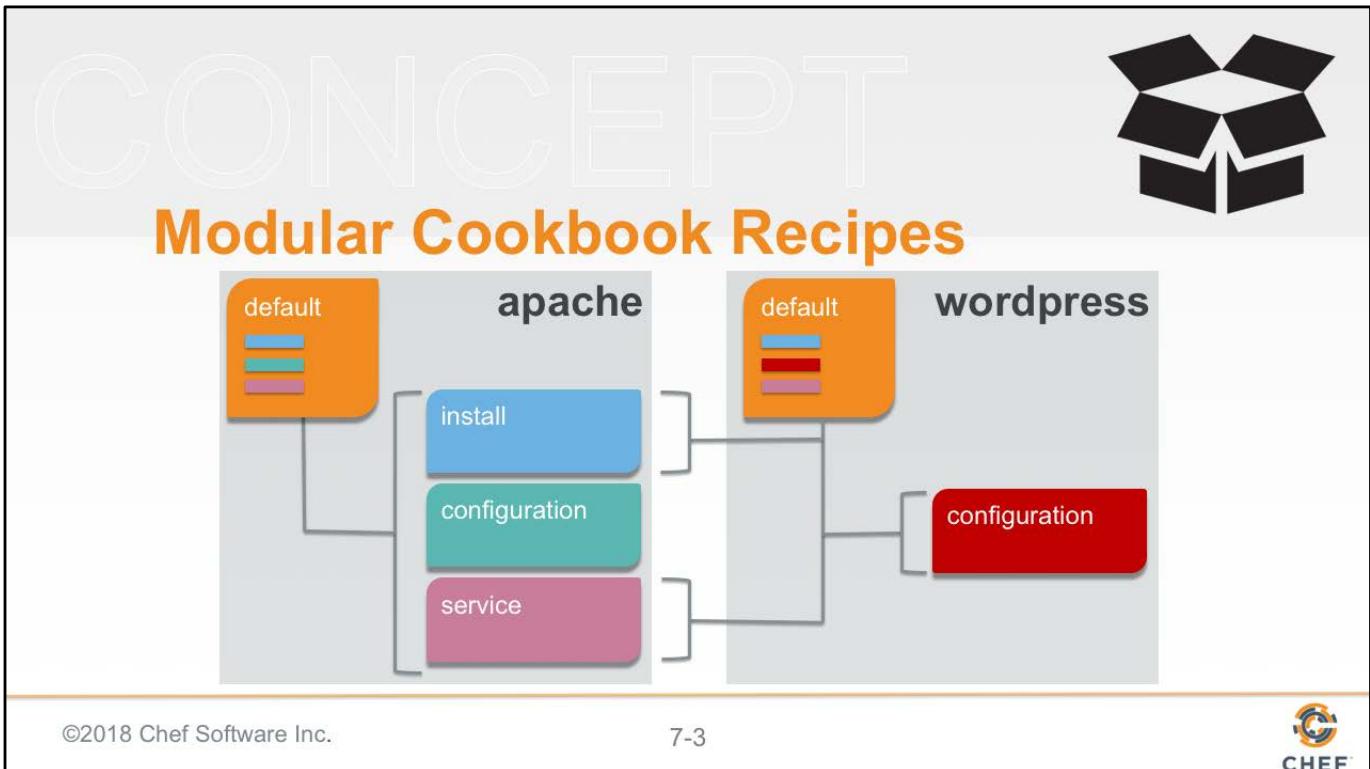
We now have the fastest feedback open source software can buy us! And right on time because it is time to refactor the cookbook again.

# Objectives

After completing this module, you should be able to:

- Refactor resources to use attributes
- Use Pry to explore the current state of execution
- Make changes to your recipes with confidence

In this module you will learn how to refactor a cookbook to use node attributes, employ pry to set up break points in your code, and make changes with confidence.



When we initially set out to create a cookbook that was more modular we broke the concerns of the webserver into three different recipes. This would allow an opportunity for cookbook authors within our organization re-use components of the cookbook by including only the recipes that they want.

Sometimes you do not want to re-define an entire new recipe and simply want to provide a different name or version for the package; a single file path for the configuration file.

# CONCEPT

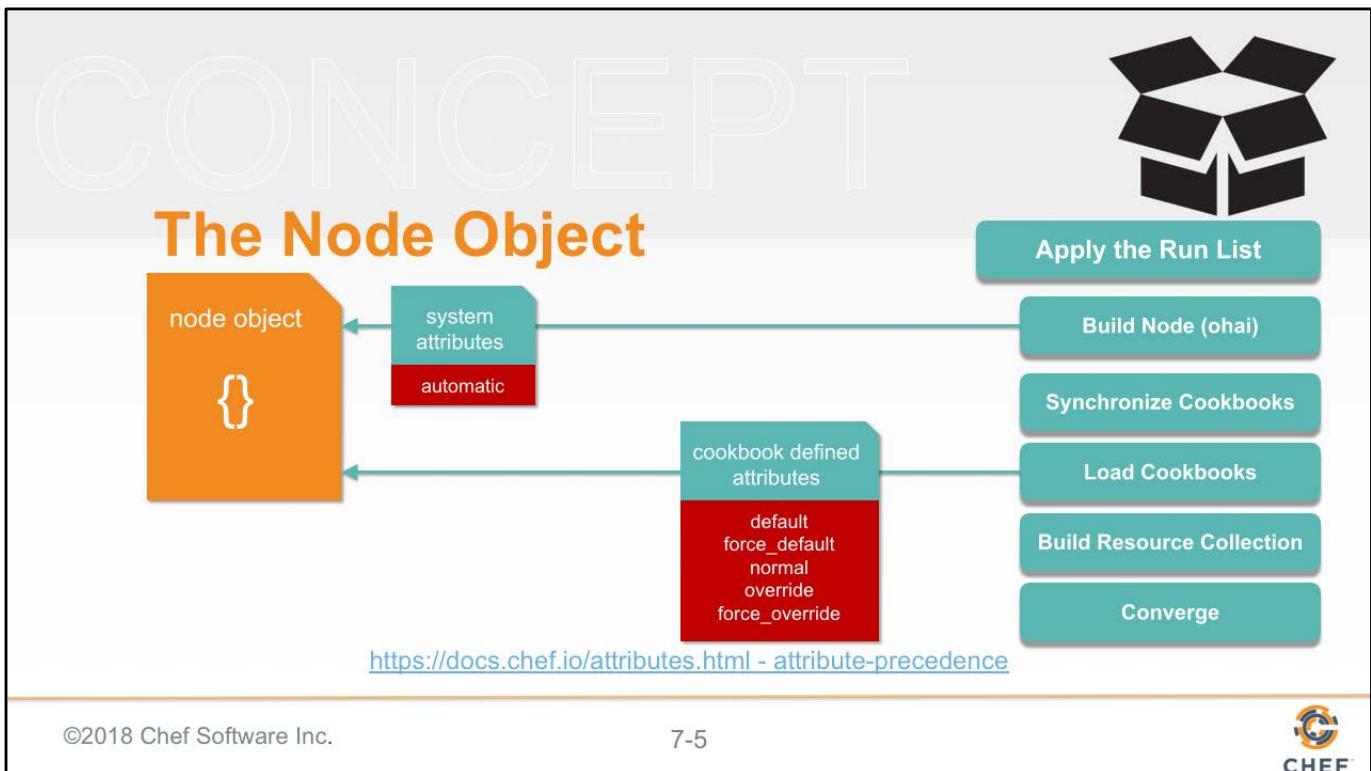
## Modular Cookbook Recipes

```
~/apache/recipes/configuration.rb
file '/var/www/html/index.html' do
  content '<h1>Welcome Home</h1>'
end
```

©2018 Chef Software Inc. 7-4 

Within each recipe we defined the resources necessary to bring the webserver into the desired state. When we expressed these resources we did so with values that worked for this platform and version of the Operating System (OS).

The configuration recipe defined a file resource with a path to the location for the default HTML page. This path is hard-coded for this particular platform. If we had a situation where another cookbook or environment or role wanted to use this recipe but provide a custom value we could not do that unless we talk about making the file path a node attribute.



Cookbooks can define node attributes which are added to the node object after the initial discovery is done by Ohai. Ohai attributes are considered automatic and cannot be overwritten. However, the attributes defined in a cookbook can come in variety of levels. This allows for cookbooks to define a base value which another cookbook can replace when needed.

That is the kind of flexibility that we want to implement in our cookbook.

# EXERCISE



## Refactor to Use Attributes

*Time to remove all the hard-coded values and make them attributes.*

**Objective:**

- Refactor the Install recipe to use a Node attribute
- Execute the tests and verify the tests fail
- Create the attributes file and add the Node attribute
- Execute the tests and verify the tests pass

Together we will walk through refactoring the install recipe continuing to use our tests to prove that we have not caused a regression in recipes.

## Replace the Value with a Node Attribute

```
~/apache/recipes/install.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: install  
#  
# Copyright:: 2018, The Authors, All Rights Reserved.  
package node['apache']['package_name']
```

Because we have expectations in place we can start with a change to the install recipe. Here we are replacing the package name with a node attribute that we have yet to define in the attributes file.

# EXERCISE



## Refactor to Use Attributes

*A change means a chance for us to run the tests!*

**Objective:**

- ✓ Refactor the Install recipe to use a Node attribute
- Execute the tests and verify the tests fail
- Create the attributes file and add the Node attribute
- Execute the tests and verify the tests pass

We made a change. Before we define the node attribute we should run the tests.

## Execute the Tests to See it Fail



```
> chef exec rspec
```

```
..FFFFF...
```

```
Failures:
```

```
1) apache::default When all attributes are default, on an Centos  
6.9 converges successfully
```

```
Failure/Error: expect { chef_run }.to_not raise_error
```

```
expected no Exception, got #<NoMethodError: undefined  
method `[]' for nil:NilClass> with backtrace:
```

```
# /tmp/chefspec20180313-21260-
```

When executing 'rspec' against all the examples that we have defined we see a large number of failures. The failure summary will show us that the chef run failed with an error. This error is informing us that we attempted to retrieve an attribute from the node object that does not exist. All of the failures should be the same.

# EXERCISE



## Refactor to Use Attributes

*We definitely broke it! Now, let's fix it.*

**Objective:**

- ✓ Refactor the Install recipe to use a Node attribute
- ✓ Execute the tests and verify the tests fail
- ❑ Create the attributes file and add the Node attribute
- ❑ Execute the tests and verify the tests pass

Now it is time to create the attributes file and define the necessary attribute.

## Ask Chef How to Generate an Attributes File



```
> chef generate attribute --help
```

```
Usage: chef generate attribute [path/to/cookbook] NAME [options]
      -C, --copyright COPYRIGHT           Name of the copyright hol...
      -m, --email EMAIL                  Email address of the auth...
      -a, --generator-arg KEY=VALUE     Use to set arbitrary ...
      -I, --license LICENSE             all_rights, apache2, mit, ...
      -g GENERATOR_COOKBOOK_PATH,       Use GENERATOR_COOKBOOK_PA...
      --generator-cookbook
```

The 'chef' tool is able to generate attributes. All it requires is the name of the file when you are inside the cookbook. We are currently inside the cookbook directory so now we need to give it a name.

## Use Chef to Generate a Default Attributes File



```
> chef generate attribute default
```

```
Compiling Cookbooks...
Recipe: code_generator::attribute
  * directory[/home/chef/apache/attributes] action create
    - create new directory /home/chef/apache/attributes
  * template[/home/chef/apache/attributes/default.rb] action create
    - create new file /home/chef/apache/attributes/default.rb
    - update content in file
/home/chef/apache/attributes/default.rb from none to e3b0c4
  (diff output suppressed by config)
```

The standard name for the attributes file is 'default'. This command will generate an attributes file named 'default.rb' in the attributes directory.

## View the Attributes File Generated



```
> tree attributes
```

```
attributes
```

```
└── default.rb
```

```
0 directories, 1 file
```

We can verify that by looking in the attributes directory to see the file has been generated.

## Add the Default Node Attribute

```
~/apache/attributes/default.rb
```

```
default['apache']['package_name'] = 'httpd'
```

Now it is time to edit the attributes file and define the node attribute. Here we are defining the node attribute at the default level. Setting it to default will allow other cookbooks to override it if necessary.

# EXERCISE



## Refactor to Use Attributes

*The work is done. Let's hope it's the right work. Run the tests!*

**Objective:**

- ✓ Refactor the Install recipe to use a Node attribute
- ✓ Execute the tests and verify the tests fail
- ✓ Create the attributes file and add the Node attribute
- Execute the tests and verify the tests pass

This change should fix all the examples that we broke when we used the node attribute without having defined it.

## Execute the Tests to See it Pass



```
> chef exec rspec
```

```
.....
```

```
Finished in 4.07 seconds (files took 3.93 seconds to load)
11 examples, 0 failures
```

The results here show all the examples pass.

# EXERCISE



## Refactor to Use Attributes

*We made a change and we know it works!*

**Objective:**

- ✓ Refactor the Install recipe to use a Node attribute
- ✓ Execute the tests and verify the tests fail
- ✓ Create the attributes file and add the Node attribute
- ✓ Execute the tests and verify the tests pass

With all the expectations having been met we can confidently say that the cookbook has been refactored successfully.

# PROBLEM



## What if We Made a Typo?

While implementing the node attribute what if made a mistake?

In the process of implementing the use of the node attribute in the recipe or in the attributes file we could have made a mistake. We proved that the examples would have caught the error.

What if an error occurred and we were unable to find it? Occasionally you will implement a change wrong and then find yourself staring at the failing expectations wondering what is wrong.

## Typos Like This One Will Waste Time

```
~/apache/attributes/default.rb
```

```
default['apche']['package_name'] = 'httpd'
```

This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

# CONCEPT



## Mental Model vs Actual Model

Faster feedback helps us build a greater mental model of the actual execution model. Tests that we define help strengthen it. However, tests are not very interactive as they are more like experiments. What we want is the ability to pause execution and look around.

This is a situation where our mental model of the state of things is different than the actual model of execution. The benefit of tests is that it allows us to express the expectations about the model of how the execution should run. Testing is like a experiment: setup; execute; verify.

That feedback is not very interactive. There are moments where you want to be to stop the execution at a particular point and ask some questions.

# CONCEPT



## Pry a Debugger

Pry is a Ruby debugger that allows you to define break points. These breakpoints allow you to pause operation and interact with the current process being able to interrogate the current state of the system.

<http://pryrepl.org>

This situation is one in which we want to use a tool called a debugger. Debuggers allow us to set up points where the execution flow will pause and allow us, the user, to interact with the system within the current context of where the execution paused.

Ruby has a well supported debugger project named 'Pry'. 'Pry' is a Ruby gem that is already installed in the Chef Development Kit (Chef DK).

# EXERCISE



## Setup a Break Point

*Time to make trouble for ourselves.*

**Objective:**

- Mutate the code and add the breakpoint
- Execute the tests to cause the breakpoint to trigger
- Remove the breakpoint and restore the code

To explore using Pry we need to create an issue for ourselves to troubleshoot. Doing so will allow us to see some of the power of Pry.

## Create a Typo in the Defined Attribute

```
~/apache/attributes/default.rb  
default['apche']['package_name'] = 'httpd'
```

This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

## Add a Break Point in the Recipe

```
~/apache/recipes/install.rb

#
# Cookbook:: apache
# Recipe:: install
#
# Copyright:: 2018, The Authors, All Rights Reserved.
require 'pry'
binding.pry

package node['apache']['package_name']
```

To use Pry you first have to specify a require statement. The require here will look for a file name 'pry' give up on finding it locally and then look for the file inside all of the installed gems.

After the Pry code is loaded we access a method named 'binding' and then ask it to run 'pry'. 'binding' is a special method in Ruby that is like gaining access to the DNA of the current context. Pry, after it is loaded, will add the 'pry' method to the binding object to allow us the ability to setup a break point.

Wherever we want to set a breakpoint we can place these two lines.

# EXERCISE



## Setup a Break Point

*Time to pry into the code and see what it is going on.*

**Objective:**

- Mutate the code and add the breakpoint
- Execute the tests to cause the breakpoint to trigger
- Remove the breakpoint and restore the code

The breakpoint cannot break itself. We need to execute the code to cause the execution to pause. The best way to do that is execute the tests that we have defined.

## Execute the Test to Initiate Pry



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

```
From: /tmp/chefspec20180313-23174-grz9vbfile_cache_path/cookbooks/apache/recipes/install.rb @ line 9
Chef::Mixin::FromFile#from_file:
```

```
4: #
5: # Copyright:: 2018, The Authors, All Rights Reserved.
6: require 'pry'
7: binding.pry
8:
=> 9: package node['apache']['package_name']
```

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

We can execute the tests for the install specification so that it will process that recipe. After a moment of normal execution the flow will pause and you will be shown where in the code the execution has paused. Along the top is the name of the file with the line number where it is paused. Below is a source code listing line-by-line before and after the breakpoint.

## Pry Provides an Interactive Prompt



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

```
# ... CONTINUED FROM THE PREVIOUS SLIDE ...
4: #
5: # Copyright:: 2018, The Authors, All Rights Reserved.
6: require 'pry'
7: binding.pry
8:
=> 9: package node['apache']['package_name']
```

```
[1] pry(<Chef::Recipe>)>
```

Below the summary of the code around the breakpoint is a prompt. Pry launches a Read-Eval-Print-Loop (REPL). At this prompt we can type in a number of commands and any Ruby code.

## Ask Pry for Help



```
[1] pry(#<Chef::Recipe>) > help
```

```
Help
  help          Show a list of commands or information about a specific command.

Context
  cd            Move into a new context (object or scope).
  find-method   Recursively search for a method within a class/module or the curr...
  ls             Show the list of vars and methods in the current scope.
  pry-backtrace Show the backtrace for the pry session.
  raise-up      Raise an exception out of the current pry instance.
  reset         Reset the repl to a clean state.
```

To escape the help menu, type in q

The most important provided by Pry is probably the 'help' command. Within the results of this you will see all the commands available. The help will display in a scrolling page like a Linux man page. To escape out of the help output and return to being able to type in commands you will need to enter the keystrokes ':q'

Instructor Note: This content introduces Pry but will not go into explaining all of the different features.

## Execute Any Code As You Would in a Recipe



```
[2] pry(#<Chef::Recipe>) > node['apache']
```

```
=> nil
```

Back at the prompt you can enter in any code that you would normally write within the recipe. In this case we can start to examine the node object to see that the node object does not have the top-level attribute set as we expected.

## Explore the Different Node Attributes



```
[3] pry(#<Chef::Recipe>) > node['apache']
```

```
=> {"package_name"=>"httpd"}
```

This interactive session allows us to verify the actual state quickly. When it does not match our mental model we can try multiple hypotheses quickly. Here we may return back to the attribute file and copy the text within the attribute and attempt this again and see what is actually going on.

We see in this example that

## Halt the Execution of the Test Immediately



```
[4] pry(#<Chef::Recipe>) > exit!
```

When you are satisfied with what you have discovered it is time to exit. Pry provides two versions of exit:

'exit' which will resume the execution and stop at any other breakpoints along the way.

'exit!' which halts the execution immediately and returns you to your shell.

In this situation we want to halt the execution immediately as we have discovered the issue.

# EXERCISE



## Setup a Break Point

*Time to pry into the code and see what it is going on.*

**Objective:**

- Mutate the code and add the breakpoint
- Execute the tests to cause the breakpoint to trigger
- Remove the breakpoint and restore the code

Now that we have discovered the issue in this scenario it is time to remove the breakpoint and restore the attributes code back to its correct state.

## Remove the Break Point from the Recipe

```
~/apache/recipes/install.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: install  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
require 'pry'  
binding.pry  
  
package node['apache']['package_name']
```

## Fix the Change in the Attributes



~/apache/attributes/default.rb

```
default['apache']['package_name'] = 'httpd'
```

+

This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

# EXERCISE



## Setup a Break Point

*Time to pry into the code and see what it is going on.*

**Objective:**

- ✓ Mutate the code and add the breakpoint
- ✓ Execute the tests to cause the breakpoint to trigger
- ✓ Remove the breakpoint and restore the code

This small exercise focused on a small subset of what is possible with Pry. It is a powerful tool that will aid you in understand the execution of the system much faster than tests alone.

# LAB



## Refactor Remaining Resources

- Refactor the resource to use a Node attribute
- Execute the tests and verify the tests fail
- Add the new Node attribute
- Execute the tests and verify the tests pass

*BONUS: Use pry to verify that the attribute has been set.*

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

Now it is your turn. Two recipes remain that I want you to refactor to use attributes. Follow the same workflow you used here. As a bonus try using Pry again to reinforce setting it up and navigating through the execution flow with it.

Instructor Note: Allow 10 minutes to complete this exercise

## Update the Recipe to use the Node Attribute

```
~/apache/recipes/service.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: service  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
service node['apache']['service_name'] do +  
  action [:enable, :start]  
end
```

Let's review the refactoring of the service resource. You returned first to the service resource in the service recipe and specify a node attribute that will give you the service name.

## Execute the Tests to See it Fail



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

```
FFF
```

```
Failures:
```

```
  1) apache::service When all attributes are default, on an Centos 6.9  
converges successfully
```

```
    Failure/Error: expect { chef_run }.to_not raise_error
```

```
      expected no Exception, got #<ArgumentError: You must supply a name when  
declaring a service resource> with backtrace:
```

```
      # /tmp/chefspec20180313-17746-  
14gwtwpfile_cache_path/cookbooks/apache/recipes/service.rb:6:in `from_file'
```

You executed the tests against all the recipes or the specific service recipe. A large set of errors appear as we saw last time. The error is telling us to define the node attribute.

## Add the Default Node Attribute



~/apache/attributes/default.rb

```
default['apache']['package_name'] = 'httpd'  
default['apache']['service_name'] = 'httpd'
```

You opened the default attributes file up and defined the new node attribute at the default level.

## Execute the Tests to See it Pass



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

```
...
```

```
Finished in 1.06 seconds (files took 4.33 seconds to load)
3 examples, 0 failures
```

You executed the tests again and saw all the expectation have been met successfully.

## Update the Recipe to use the Node Attribute

```
~/apache/recipes/configuration.rb

#
# Cookbook:: apache
# Recipe:: configuration
#
# Copyright:: 2018, The Authors, All Rights Reserved.
file node['apache']['default_index_html'] do
  content '<h1>Welcome Home!</h1>'
end
```

Let's review the refactoring of the service resource. You returned first to the service resource in the service recipe and specify a node attribute that will give you the service name.

## Execute the Tests to See it Fail



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

```
FF
```

```
Failures:
```

```
  1) apache::configuration When all attributes are default, on an Centos 6.9  
conve
```

```
    Failure/Error: expect { chef_run }.to_not raise_error
```

```
      expected no Exception, got #<ArgumentError: You must supply a name when  
decurce> with backtrace:
```

You executed the tests against all the recipes or the specific service recipe. A large set of errors appear as we saw last time. The error is telling us to define the node attribute.

## Add the Default Node Attribute



~/apache/attributes/default.rb

```
default['apache']['package_name'] = 'httpd'  
default['apache']['service_name'] = 'httpd'  
default['apache']['default_index_html'] = '/var/www/html/index.html'
```

You opened the default attributes file up and defined the new node attribute at the default level.

## Execute the Tests to See it Pass



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

```
..
```

```
Finished in 2.27 seconds (files took 1.82 seconds to load)
2 examples, 0 failures
```

You executed the tests again and saw all the expectation have been met successfully.

# LAB



## Refactor Remaining Resources

- ✓ Refactor the resource to use a Node attribute
- ✓ Execute the tests and verify the tests fail
- ✓ Add the new Node attribute
- ✓ Execute the tests and verify the tests pass

*BONUS: Use pry to verify that the attribute has been set.*

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

Congratulations. Now you have completely refactored the resources in the cookbook to use node attributes.

Let's have a discussion.

Instructor Note: We did not review the configuration recipe

# DISCUSSION



## Discussion

What are the benefits of providing the package name and service name as node attributes?

What value does Pry provide to you as a Cookbook Developer?

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?

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**

With the resources now using node attributes we are ready to explore the last section which will challenge us to expand the scope of this cookbook to support multiple platforms.



## Testing While Refactoring to Multiple Platforms



©2018 Chef Software Inc.

8-1



When we started developing this cookbook I told you that we were going to continue to refactor this cookbook until it supported multiple platforms. We could have started with that goal. Instead we started small. One test. One recipe. Refactor. Add more tests. Refactor. This process allowed us to deliver a reliable cookbook in a confident way. But testing was not the only thing that aided us in building this cookbook.

Instrumental to software development and test-driven development is learning how to divide the work into these small increments. Small, deliverable, verifiable steps are essential to developing code with confidence. Now that you have seen and experienced the Test Driven Development (TDD) workflow and understand the basics, the real work that lies before you is to understand how to find these divisions in the requirements you are given.

This was a hand-picked experience. The moves we made may have seemed contrived. As with any knowledge transfer the best we can do is give you a model to play with and hope the forms hold true when it comes time for you to solve a problem with real requirements.

# Objectives

After completing this module, you should be able to:

- Define expectations for multiple platforms
- Implement a cookbook that supports multiple platforms

In this module you will learn how to define expectations for multiple platforms and implement a cookbook that supports multiple platforms.

# EXERCISE



## Node Platform in ChefSpec

*What platform is the node when running a ChefSpec test?*

*How might you find out what is the platform?*

### Objective:

- Insert a break point, execute the tests, and determine the node's platform
- Remove the break point and transcend documentation

Then you will bridge the gap!



In this module we are going to develop solution in the opposite of the way we started. Instead of approaching this problem from the outside-in we are going to build it inside-out.

To do that means we are going to leverage the specifications we have written that validate the resources within our recipe. But before we do we need to gather some information that is important. Like the name of the platform we are using?

We could attempt to solve this problem by looking for documentation or a general search on the Internet. Instead we will ask the one source that knows the best: the executing code itself.

## Add a Break Point to the Default Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
require 'pry'  
binding.pry  
include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
include_recipe 'apache::service'
```

## Execute the Tests to Initiate Pry



```
> chef exec rspec
```

```
From: /tmp/chefspec20180313-24027-408ikaf file_cache_path/cookbooks/apache/recipes/default.rb @ line 8
Chef::Mixin::FromFile#from_file:

  3: # Recipe:: default
  4: #
  5: # Copyright:: 2018, The Authors, All Rights Reserved.
  6: require 'pry'
  7: binding.pry
=> 8: include_recipe 'apache::install'
  9: include_recipe 'apache::configuration'
10: include_recipe 'apache::service'
```

Execute the tests.

## Query the Node Object's Platform



```
[1] pry(#<Chef::Recipe>) > node['platform']
```

```
=> "centos"
```

And then query the platform of the node object. The results should tell you that the platform for the node object in the ChefSpec environment is 'chefspec'.

# REFERENCE



## Fauxhai

ChefSpec uses the platform you specify in the runner. You can specify any platform from the list of platforms that are stored in a gem named 'Fauxhai'.

The gem contains static node objects for most major platforms and versions.

<https://github.com/customink/fauxhai/tree/master/lib/fauxhai/platforms>

<https://github.com/customink/fauxhai>

The 'chefspec' platform is set by the ChefSpec gem. The platform has gone unspecified and this is what ChefSpec defaults to use. Now that we care about the platform we need to learn about another gem named Fauxhai. ChefSpec employs Fauxhai to provide fake node object data for various platforms.

These platforms and their various versions are defined in the gem itself. Essentially the gem, at the time of writing this, contains a large number of JSON files which hold the node object results on each specific platform and version it supports. The best way to learn what platforms are provided is to read the source code in the Fauxhai repository.

## Immediately Exit the Execution



```
[2] pry(#<Chef::Recipe>) > exit!
```

Now that we know the platform it is time to exit the execution.

# EXERCISE



## Node Platform in ChefSpec

*What platform is the node when running a ChefSpec test?*

*How might you find out what is the platform?*

### Objective:

- Insert a break point, execute the tests, and determine the node's platform
- Remove the break point and transcend documentation

A tidy life is a healthy life.



Using Pry we were able to learn something about the system without having to rely on documentation. To understand the available platforms you have to rely on reading the source code.

Learning this powerful skill of gathering details will help you solve mysteries and provide more details and queries when searching for help on the Internet. The better you can get at understanding when to employ Pry and how to use it will eventually have you using documentation less and using executing code and source code more.

Instructor Note: Finding out which platforms and versions ChefSpec supported alluded me when first working with the project. There is some mention in the ChefSpec README but I believe I found myself diving into source code and stumbling upon the Fauxhai code. This is something that would be great to show to show learners if you are capable of figuring that out.

## Remove the Break Point from the Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
  
# Copyright:: 2018, The Authors, All Rights Reserved.  
require 'pry'  
binding.pry  
include_recipe 'apache::install'  
include_recipe 'apache::configuration'  
include_recipe 'apache::service'
```

It is a good habit to clean up this break points. Leaving them around has a nasty habit of pausing the execution of a run you want to see complete uninterrupted.

# EXERCISE



## Node Platform in ChefSpec

*What platform is the node when running a ChefSpec test?*

*How might you find out what is the platform?*

### Objective:

- ✓ Insert a break point, execute the tests, and determine the node's platform
- ✓ Remove the break point and transcode documentation

Now I am ready to be shaved.



# EXERCISE



## Support for CentOS & Ubuntu

*The best of both worlds!*

**Objective:**

- Write a test that verifies the Install recipe chooses the correct package on CentOS & Ubuntu
- Execute the tests and verify the tests fail
- Update the attribute to provide support for CentOS & Ubuntu
- Execute the tests and verify the tests pass

Together let's walk through refactoring the cookbook's install recipe. Like we have done before. When we are done it will be your turn to implement the solution for the remaining recipes.

## Add a Second Context for Another Platform

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

# ... REST OF SPEC FILE ...
context 'When all attributes are default, on Ubuntu 14.04' do
  let(:chef_run) do
    runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
    runner.converge(described_recipe)
  end

  it 'converges successfully' do
    expect(chef_run).to_not raise_error
  end

  it 'installs the necessary package' do
    expect(chef_run).to install_package('apache2')
  end
end
end
```

Now return to the specification file and alongside CentOS example group it is time to define the example group that will contain the examples for the Ubuntu 14.04 platform.

The format is nearly identical between these two example groups save for the context, the parameters specified to the ServerRunner initialization, and the name of the necessary package to install.

# EXERCISE



## Support for CentOS & Ubuntu

*Seems like a lot of duplication but its worth it for the test coverage.*

**Objective:**

- ✓ Write a test that verifies the Install recipe chooses the correct package on CentOS & Ubuntu
- Execute the tests and verify the tests fail
- Update the attribute to provide support for CentOS & Ubuntu
- Execute the tests and verify the tests pass

The examples have now been defined for the existing platform and the new platform.

## Execute the Tests to See it Fail



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

```
...F
```

```
Failures:
```

```
1) apache::install When all attributes are default, on Ubuntu 14.04 installs  
the necessary package
```

```
Failure/Error: expect(chef_run).to install_package('apache2')
```

```
expected "package[apache2]" with action :install to be in Chef run.  
Other package resources:
```

```
apt_package[httpd]
```

When it comes time to execute the tests we should see that defining the new platform will not raise an error when it converges but will fail to meet the expectation that we installed the correctly named package.

# EXERCISE



## Support for CentOS & Ubuntu

*Failure means we have work to do!*

**Objective:**

- ✓ Write a test that verifies the Install recipe chooses the correct package on CentOS & Ubuntu
- ✓ Execute the tests and verify the tests fail
- Update the attribute to provide support for CentOS & Ubuntu
- Execute the tests and verify the tests pass

The name of the package is defined in the attributes file. That is what we refactored to support in the last section. It is now time to return to the attributes file and have it specify a different package name based on the platform.

# CONCEPT



## Switching on Node Platform

To control the flow of execution we need to employ some Ruby conditional statements. Conditional statements allow us to alter this control flow. Because we have access to the power of Ruby we have many choices.

[https://docs.chef.io/dsl\\_recipe.html#sts=case%20Statements](https://docs.chef.io/dsl_recipe.html#sts=case%20Statements)

## Update the Attributes to Support Platforms

```
~/apache/attributes/default.rb

case node['platform']
when 'ubuntu'
  default['apache']['package_name'] = 'apache2'
else
  default['apache']['package_name'] = 'httpd'
end

default['apache']['package_name'] = 'httpd'
default['apache']['service_name'] = 'httpd'
default['apache']['default_index_html'] = '/var/www/html/index.html'
```

A very common way is to define a case statement. The case statement allows you to provide a value or value stored in a variable to the case keyword. Then following the case statement are a number of 'when' statements. Each 'when' needs to be provided with a value or value stored in a variable. If the value in the case statement equals the value in when statement then it is match and the flow of execution will take that path and ignore all others.

If none were to match we might be in trouble as the node attribute would never be set so we can use an 'else' statement which is as good as saying if none of those match then use this path.

The order of the case statement is particularly important as well. The first match that is made is the path the execution will take.

Instructor Note: When we say 'equal' each other we mean that Ruby is comparing the objects together with the equality method, the triple equals (==) .

# EXERCISE



## Support for CentOS & Ubuntu

*This should do it!*

**Objective:**

- ✓ Write a test that verifies the Install recipe chooses the correct package on CentOS & Ubuntu
- ✓ Execute the tests and verify the tests fail
- ✓ Update the attribute to provide support for CentOS & Ubuntu
- Execute the tests and verify the tests pass

Now that the attributes file has been updated it is time execute the tests again and see if we defined this conditional logic correctly.

## Execute the Tests to See it Pass



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

```
....
```

```
Finished in 1.35 seconds (files took 4.51 seconds to load)
```

```
4 examples, 0 failures
```

Executing the tests we should see both platforms will converge without error and install the necessary packages.

# EXERCISE



## Support for CentOS & Ubuntu

*Woot! Multi-platform support for the installation!*

**Objective:**

- ✓ Write a test that verifies the Install recipe chooses the correct package on CentOS & Ubuntu
- ✓ Execute the tests and verify the tests fail
- ✓ Update the attribute to provide support for CentOS & Ubuntu
- ✓ Execute the tests and verify the tests pass

This approach to leverage the existing examples and use them to help define new examples for a new platform allowed us to build confidence through testing from the inside-out.

Taking this inside-out approach can feel right in situations where you know the steps you have to take.

# LAB



## Support for CentOS & Ubuntu

- Write a test that verifies the Service recipe chooses the service named 'httpd' on CentOS and 'apache2' on Ubuntu
- Execute the tests and verify the tests **fail**
- Update the attribute to choose the service name 'httpd' on CentOS and 'apache2' on Ubuntu
- Execute the tests and verify the tests **pass**

Now as an exercise for you it is time to do the same thing for the service recipe. The service for Ubuntu is named 'apache2'. Start with the changes to the specifications, move through see the failure, update to use the same conditional statement structure and then see the examples verify your work.

Instructor Note: Allow 10 minutes to complete this exercise

## Add a Second Context for Another Platform

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

# ... REST OF SPEC FILE ...
context 'When all attributes are default, on Ubuntu 14.04' do
  let(:chef_run) do
    runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
    runner.converge(described_recipe)
  end
  # ... it converges successfully ...

  it 'starts the appropriate service' do
    expect(chef_run).to start_service('apache2')
  end
  it 'enables the appropriate service' do
    expect(chef_run).to enable_service('apache2')
  end
end
end
```

You now define an entire example group dedicated to the Ubuntu platform which defines the same structure of examples but with the values that are important for the platform.

## Execute the Tests to See it Fail



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

```
....FF
```

```
Failures:
```

```
1) apache::service When all attributes are default, on Ubuntu 14.04 starts  
the necessary service
```

```
Failure/Error: expect(chef_run).to start_service('apache2')
```

```
expected "service[apache2]" with action :start to be in Chef run. Other  
service resources:
```

```
service[httpd]
```

Executing the test you would see the appropriate failures for the correctly named services not being started and enabled.

## Update the Attribute to Support Platforms

```
~/apache/attributes/default.rb
```

```
case node['platform']
when 'ubuntu'
  default['apache']['package_name'] = 'apache2'
  default['apache']['service_name'] = 'apache2'
else
  default['apache']['package_name'] = 'httpd'
  default['apache']['service_name'] = 'httpd'
end

default['apache']['service_name'] = 'httpd'
default['apache']['default_index_html'] = '/var/www/html/index.html'
```

Updating the attributes for the service should be a little less work because the structure is all in place.

## Execute the Tests to See it Pass



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

```
.....
```

```
Finished in 1.84 seconds (files took 4.22 seconds to load)
```

```
6 examples, 0 failures
```

Finally when we execute the tests again we see that all the examples pass.

# LAB



## Support for CentOS & Ubuntu

- ✓ Write a test that verifies the Service recipe chooses the service named 'httpd' on CentOS and 'apache2' on Ubuntu
- ✓ Execute the tests and verify the tests **fail**
- ✓ Update the attribute to choose the service name 'httpd' on CentOS and 'apache2' on Ubuntu
- ✓ Execute the tests and verify the tests **pass**

That was nearly identical and a good way to reinforce the testing flow.

# LAB



## Support for CentOS & Ubuntu

- Write a test that verifies the file recipe chooses the same path (name)  
'/var/www/html/index.html' on CentOS and on Ubuntu
- Execute the tests that verify the tests **pass**
- Update the attribute to choose the same path on CentOS and on Ubuntu
- Execute the tests that verify the tests **pass**
- Get nervous! Mutate the attributes file!
- Undo the entire attributes change and verify the tests **pass**

This is where it all comes together.



## Add a Second Context for Another Platform

```
□ ~/spec/unit/recipes/configuration_spec.rb

# ... REST OF SPEC FILE ...

context 'When all attributes are default, on Ubuntu 14.04' do
  let(:chef_run) do
    runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
    runner.converge(described_recipe)
  end
  # ... it converges successfully ...

  it 'creates the index.html' do
    expect(chef_run).to render_file('/var/www/html/index.html').with_content('<h1>Welcome
Home!</h1>')
  end
end
end
```

You then define another example group dedicated to the Ubuntu platform. Except this time the expectation is exactly the same.

## Execute the Tests to See it Pass



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

```
....
```

```
Finished in 1.84 seconds (files took 4.22 seconds to load)
```

```
4 examples, 0 failures
```

Executing the tests shows you that everything is working.

## Update the Attribute to Support Platforms

```
~/apache/attributes/default.rb

case node['platform']
when 'ubuntu'
  default['apache']['package_name'] = 'apache2'
  default['apache']['service_name'] = 'apache2'
  default['apache']['default_index_html'] = '/var/www/html/index.html'
else
  default['apache']['package_name'] = 'httpd'
  default['apache']['service_name'] = 'httpd'
  default['apache']['default_index_html'] = '/var/www/html/index.html'
end

default['apache']['default_index_html'] = '/var/www/html/index.html'
```

You implemented the change that we have done before.

## Execute the Tests to See it Pass



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

```
....
```

```
Finished in 1.84 seconds (files took 4.22 seconds to load)
```

```
4 examples, 0 failures
```

And finally see the tests pass again. This is where you should become uncomfortable that we may have a false positive and that is a good time to ensure that you do not by mutating the code.

## Heckle the code

```
~/apache/attributes/default.rb

case node['platform']
when 'ubuntu'
  default['apache']['package_name'] = 'apache2'
  default['apache']['service_name'] = 'apache2'
  default['apache']['default_index_html'] = '/var/www/html/index.html2'
else
  default['apache']['package_name'] = 'httpd'
  default['apache']['service_name'] = 'httpd'
  default['apache']['default_index_html'] = '/var/www/html/index.html'
end
```

So anywhere in the Ubuntu flow of execution make a small mutation. In the example I am providing I have chosen a different path. Removing the attribute is another option as well.

## Execute the Tests to See it Pass



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

```
...F
```

```
Failures:
```

```
1) apache::configuration When all attributes are default, on
Ubuntu 14.04 creates the index.html
    Failure/Error: expect(chef_run).to
      render_file('/var/www/html/index.html').with_content('<h1>Welcome
      Home!</h1>')

    expected Chef run to render "/var/www/html/index.html"
```

Executing the tests should net at least one failure and that should give you more confidence that the expectations you have written are doing the work you want them to do.

## Update the Attributes

```
~/apache/attributes/default.rb

case node['platform']
when 'ubuntu'
  default['apache']['package_name'] = 'apache2'
  default['apache']['service_name'] = 'apache2'
  default['apache']['default_index_html'] = '/var/www/html/index.html2'
else
  default['apache']['package_name'] = 'httpd'
  default['apache']['service_name'] = 'httpd'
  default['apache']['default_index_html'] = '/var/www/html/index.html'
end

default['apache']['default_index_html'] = '/var/www/html/index.html'
```

Finally you might restore the code. Removing the mutation.

You may even choose to undo the change the proposed change. This is up to you to make the decision. In the example shown here I have returned to the original implementation. The original implementation worked, executing our tests proved it. Whether you should leave the attribute defined in the case statement or outside of it is up to you.

Leaving it in the case statements ensures that all values are defined on the platform. If a value on a particular platform were to change we would simply need to only change it within that platform's flow of control. However, if you never implement another platform you have created two lines of code. Some may argue the fewer lines of code you issue or statements you place inside of a conditional make it easier to read and understand.

The most important thing is that the examples you defined should remain in the specification regardless of the implementation. The examples describe the expected behavior of the platform.

## Execute the Tests to See them Pass



```
> chef exec rspec
```

```
.....
```

```
Finished in 6.02 seconds (files took 4.02 seconds to load)
```

```
18 examples, 0 failures
```

Executing the tests should net at least one failure and that should give you more confidence that the expectations you have written are doing the work you want them to do.

# LAB



## Support for CentOS & Ubuntu

- ✓ Write a test that verifies the file recipe chooses the same path (name)  
'/var/www/html/index.html' on CentOS and on Ubuntu
- ✓ Execute the tests that verify the tests **pass**
- ✓ Update the attribute to choose the same path on CentOS and on Ubuntu
- ✓ Execute the tests that verify the tests **pass**
- ✓ Get nervous! Mutate the attributes file!
- ✓ Undo the entire attributes change and verify the tests **pass**

There is only one more thing to do.



# PROBLEM



## What About an Integration Test

Remember that ChefSpec and Fauxhai are fake in-memory representations of a chef-client run. They are not equivalent to running the recipe on the specified platform.

Now that we have finished building everything from the inside-out. It is finally time to see if the integration test works. This is important. When building recipes with ChefSpec you can very quickly make mistakes. Those mistakes are not the typos or omissions we have made. These are the mistakes that only the platform can catch.

Because we have been doing everything in-memory we really do not know if the package name, file path, or service name actually works. The only way to prove that is to apply the recipe to that platform.

# EXERCISE



## Integration Test with Ubuntu

*This is where it all started.*

**Objective:**

- Update the Kitchen Configuration to test on Ubuntu
- Execute the integration tests and verify that they pass

So for our last and final exercise together lets update the Kitchen configuration to give us the ability to test on the Ubuntu platform.

## Add a New Platform to the Kitchen Configuration

```
~/apache/.kitchen.yml
```

```
---
```

```
driver:
```

```
  name: docker
```

```
provisioner:
```

```
  name: chef_zero
```

```
verifier:
```

```
  name: inspec
```

```
platforms:
```

```
  - name: centos-6.9
```

```
  - name: ubuntu-14.04
```

Within the kitchen configuration we define the new Ubuntu 14.04 platform.

## Verify the New Instance is Present



```
> kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action
default-centos-69	Docker	ChefZero	InSpec	Ssh	Verified
default-ubuntu-1404	Docker	ChefZero	InSpec	Ssh	<Not Created>

We verify that the platform exists within the list of instances.

# EXERCISE



## Integration Test with Ubuntu

*Fingers crossed*

**Objective:**

- Update the Kitchen Configuration to test on Ubuntu
- Execute the integration tests and verify that they pass

And now it is time to execute the test suite. By choosing a very valuable and implementation free InSpec example, is the website up and running in localhost, we can be fairly certain that the expectations should be met.

## Execute the Tests for All Platforms



```
> kitchen test
```

```
-----> Starting Kitchen (v1.19.1)
-----> Cleaning up any prior instances of <default-centos-69>
-----> Destroying <default-centos-69>...
      Finished destroying <default-centos-69> (0m0.00s).
-----> Testing <default-centos-69>
-----> Creating <default-centos-69>
      ...
      ...
```

To execute the tests against both platforms run 'kitchen test'. Because we have two instances and did not specify a particular instance with the command it will run tests against all the listed instances.

This might be a good time to get up and move around as it will take some time.

# EXERCISE



## Integration Test with Ubuntu

*Now I'm sure the cookbook works on two platforms and it would be easy to add a third ... or fourth.*

**Objective:**

- ✓ Update the Kitchen Configuration to test on Ubuntu
- ✓ Execute the integration tests and verify that they pass

Your work has only begun



The expectations should pass and this brings the last exercise to a close.

Let's have a discussion.

# DISCUSSION



## Discussion

What are the benefits and drawbacks of defining unit tests for multiple platforms?

What are the benefits and drawbacks of defining integration tests for multiple platforms?

When testing multiple platforms would you start with integration tests or unit tests?

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?

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



Thank you for your time and attention.



## Approaches to Extending Resources

You express the state of your infrastructure with resources, defined in recipes, encapsulated in cookbooks. Chef provides a core set of resources (dependent on your version of Chef and your platform). These core resources allow you to express the desired state of your infrastructure in a majority of situations. They can also be combined together to express the desired state where these individual resources fall short.

Early on when working with Chef these core resources and their ability to be combined will handle a majority of the configuration management issues that you face. After awhile you will come across more specific resource needs that have not yet been created or perhaps help describe a common set of resources you continue to use together.

When a necessary resource does not exist or when you want to express a group of resources a single resource, Chef provides a few ways to accomplish this.

# Objectives

After completing this module, you should be able to:

- Describe the difference between:
  - Custom Resources
  - Definitions
  - Heavy-Weight Resource-Providers
  - Light-Weight Resource-Providers

After completing this module you will be able to describe the differences between Custom Resources, Definitions, Heavy-Weight Resource Providers and Light-Weight Resource Providers.

# CONCEPT

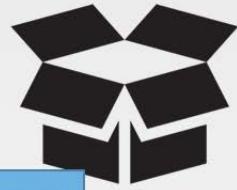


## Approaches to Extending Resources

- 1 Pure Ruby (Heavy-Weight Resource-Providers / HWRP)
- 2 Definitions
- 3 Light-Weight Resource-Providers (LWRP)
- 4 Custom Resources

Having reached the limit of the core set of resources presents a new set of challenges before you. Fortunately these challenges are not insurmountable because of some of the design choices Chef has made to make it possible to extend its functionality. Chef is a maturing product that continues to evolve to bring joy to its users. While we are going to focus on Custom Resources it is important that have a basic understanding of these other implementations.

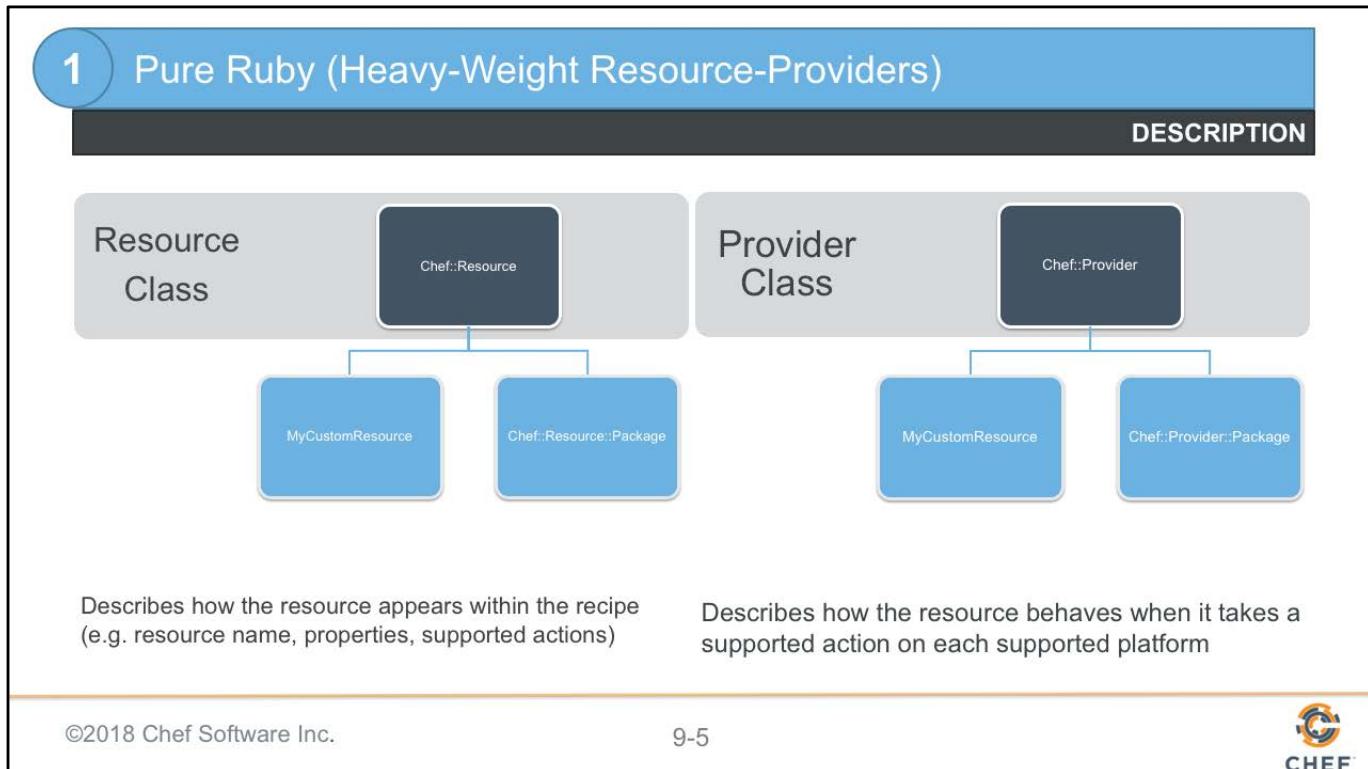
# CONCEPT



## 1 Pure Ruby (Heavy-Weight Resource-Providers / HWRP)

- Description
- File and Folder Structure
- Implementation Language & Usage
- Benefits & Drawbacks

I will provide a description of each, explain the files and folder structure, take a quick look at how each is implemented, and then talk about any requirements or limitations when pursuing this implementation choice.



Chef's core resources are written in Ruby. The first approach to creating your own resources is to create your own with Ruby classes. These pure Ruby implementations of Resources is often referred to as Heavy-Weight Resource-Provider, or HWRP. Each resource defined in Chef is defined in two classes which sub-class the core Chef Resource and Chef Provider class. Sub-classing is an object-oriented programming term that means to inherit characteristics (e.g. methods and variables) from the parent class. Within the subclass you are required to override specific methods for the class to behave as a resource within the system.

The Chef::Resource class describes how the resource appears within the recipe; the interface. The Chef::Provider class describes how the resource will act when it takes one of the supported action on each supported platform.

## 1 Pure Ruby (Heavy-Weight Resource-Providers)

### STRUCTURE

#### my\_cookbook

```
libraries/
• [my_custom_resource]_resource.rb
• [my_custom_resource]_provider.rb
```

They are stored within the libraries folder in separate files for the resource and the provider. The file names are snake case representations of the class name stored within the file.

An HWRP, as pure Ruby, is stored in within the 'libraries' directory. Each class, one for the resource and the provider, are stored in separate files. The name of the file matches the class name except it has been snake-cased. Snake-casing lower cases the class name and places underscores between letters where capital letters used to exist. This is a common Ruby practice and one enforced by Rubocop. All the Ruby files within that directory are evaluated after the cookbook is synchronized and loaded.

## 1 Pure Ruby (Heavy-Weight Resource-Providers)

### IMPLEMENTATION LANGUAGE - RESOURCE

libraries/apache\_vhost\_resource.rb

```
class Chef
  class Resource
    class ApacheVhost < Chef::Resource
      def initialize(name, run_context=nil)
        super
        @resource_name = :apache_vhost          # Defining the resource name
        @provider = Chef::Provider::ApacheVhost # Specifying which Provider to use
        @action = :create                      # Setting the default action
        @allowed_actions = [:create, :remove]   # Setting the list of actions
        # ... SETUP ANY DEFAULT VALUES HERE ...
      end

      def site_name(arg=nil)
        set_or_return(:site_name, arg, :kind_of => String)
      end
    end
  end
end
```

When defining the resource for a Heavy-Weight Resource-Provider you sub-class the Chef Resource class. The initialize method is overridden to specify new default values and allows us to configure the class as necessary when the resource is created in memory. Each potential attribute is defined as a method which uses a helper to setup the default values, value types it supports, etc.

## 1 Pure Ruby (Heavy-Weight Resource-Providers)

### IMPLEMENTATION LANGUAGE - PROVIDER

libraries/apache\_vhost\_provider.rb

```
class Chef
  class Provider
    class ApacheVhost < Chef::Provider
      def load_current_resource
        @current_resource ||= Chef::Resource::ApacheVhost.new(new_resource.name)

        @current_resource.site_name(new_resource.site_name)
        # ... remaining properties defined in the resource
        @current_resource
      end

      def action_create
        # ... code that creates the resource on all supported platforms ...
      end
    end
  end
end
```

When defining the provider for a Heavy-Weight Resource-Provider you sub-class the Chef Provider class. The initialize method does not have to be overridden. The load\_current\_resource method must be overridden and is where the configuration from the resource is created and configured for use in each of the supported actions. The actions here are defined as methods with the prefix 'action\_' and within them you would define the code necessary to perform the operations for this resource.

Chef provides additional helpers to allow you to shell out to perform operations on the system. You also have the entire Ruby language and any gems that might be packaged with the Chef DK (or you have added to Chef DK) at your disposal.

## 1 Pure Ruby (Heavy-Weight Resource-Providers)

### USAGE

recipes/default.rb

```
apache_vhost 'welcome' do
  action :delete
end

apache_vhost 'users'

apache_vhost 'admins' do
  site_port 8080
end
```

The resource would now be available within any recipe defined in this cookbook or any cookbook that adds this cookbook as a dependency. Here in this example recipe the resources delete and creates apache sites. All three of the resources rely on the site name attribute being tied to the name provided to the resource. The first deletes the welcome site. The next two both rely on the default action of create. The second resource assumes the default site port value.

## 1 Pure Ruby (Heavy-Weight Resource-Providers)

### BENEFITS & DRAWBACKS

- Available in some of the earliest versions of Chef
- Allows for extremely flexible and powerful resource implementations
- Requires knowledge of Ruby
- Requires knowledge of Object-Oriented Programming techniques

HWRP are incredibly useful when you need the full power of Ruby to implement your own resource. However, they come at the cost of understanding a number of Object-Oriented Programming techniques and the Ruby language. When exploring community cookbooks you may find examples of these resources in use.

## 2 Definitions

### DESCRIPTION

```
recipes/admins_site.rb
```

```
apache_vhost 'admins' do
  site_name 'admins'
end
```

```
recipes/users_site.rb
```

```
apache_vhost 'users' do
  site_name 'users'
end
```

```
recipes/dogs_site.rb
```

```
...
```

```
definitions/apache_vhost.rb
```

```
define :apache_vhost site_name: 'default' do
  directory ...
  template ...
  file ...
end
```

Definitions behaves like a compile time macro that is reusable across recipes. Macros allow you to write a small amount of code that expands out into the contents of the definition wherever it is found within the recipes. With a definition you give it a name, provide parameters, and specify a block of code.

## 2 Definitions

### STRUCTURE

#### my\_cookbook

```
definitions/
• [my_definition_name].rb
```

They are stored within the definitions folder and often the name of the definition defines of the file.

The code that defines the definition is stored within a definitions directory in a Ruby file that is processed with the definition Domain Specific Language.

## 2 Definitions

### IMPLEMENTATION LANGUAGE

```
definitions/apache_vhost.rb
```

```
define :apache_vhost site_name: 'default', site_port: 80 do
  directory "/srv/apache/#{params[:site_name]}/html" do
    recursive true
    mode '0755'
  end

  templates "/srv/apache/#{params[:site_name]}/html" do
    source 'conf.erb'
    mode '0644'
    variables(document_root: "/srv/apache/#{params[:site_name]}/html", port: params[:site_port])
    mode '0755'
    notifies :restart, 'service[httpd]'
  end

  # ... remaining resources ...
end
```

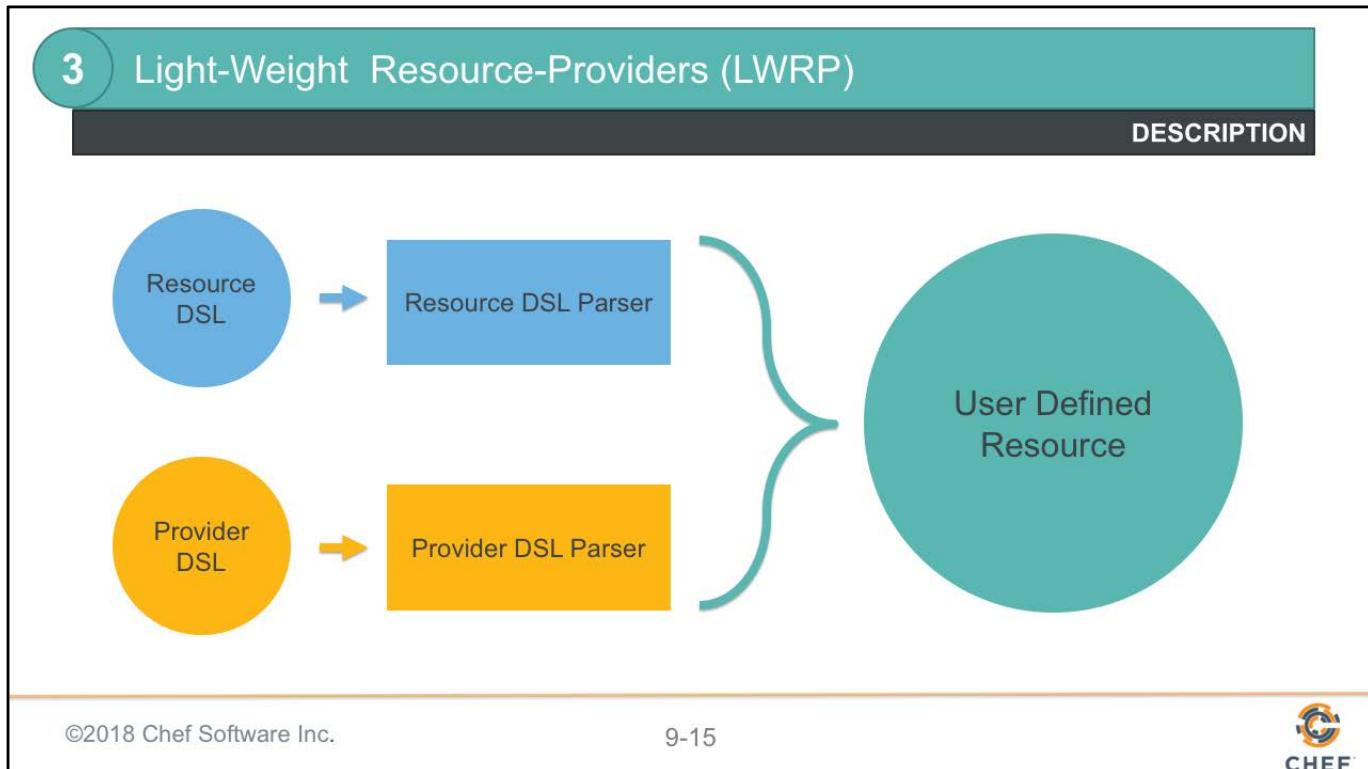
When creating a definition you specify a name and a hash of any parameters you wish to provide. Within the definition the parameters are retrievable from a hash named `params`. The use of the definition within a recipe looks similar to a resource but that is not the case. Definitions cannot notify other resources, subscribe to notifications from other resources, (i.e. `notifies` and `subscribes`) and cannot employ guards (i.e. `only_if` and `not_if`).

## 2 Definitions

### BENEFITS & DRAWBACKS

- Available in some of the earliest versions of Chef
- Allows for code re-use within recipes
- Definition usage could be mistaken for a true resource
- Definitions do not support notifications (`subscribes` and `notifies`)

Definitions shipped in some of the earliest versions of Chef and are still supported today. However, as of Chef 12.5 it is strongly recommended that you choose a solution built with custom resources.



Light-Weight Resource-Provider, or LWRP, are Chef resources defined in two Domain Specific Languages (DSL) that allow you to create resources without having to understand the complexity presented by HWRP.

An LWRP is as much a resource as the core resources defined in Chef. The resource and the provider is parsed and converted into Ruby objects.

### 3 Light-Weight Resource-Providers (LWRP)

#### STRUCTURE

## my\_cookbook

```
resources/
  • [my_resource_name].rb
providers/
  • [my_resource_name].rb
```

An LWRP is defined in two separate files that share the same name. The resource definition is defined in the resources directory of the cookbook; the provider definition in the providers directory.

The cookbook name is combined with the file name to create the name of the resource.

A single LWRP definition is defined in two separate files. The file is named exactly the same but one file resides in the 'resources' directory; the other in the 'providers' directory. Both of these files are parsed after the cookbook is synchronized and loaded. Each file's DSL is then converted into Ruby class at runtime.

Within the file in the 'resources' directory you define the interface for the custom resource. There, within a resource DSL, you can specify a name of the resource, the list of available actions, the default action, and the properties that may be set for the resource. Within the file in the 'providers' directory you define the implementation for the custom resource. There, within a provider DSL, you specify what happens when an action is chosen.

### 3 Light-Weight Resource-Providers (LWRP)

#### IMPLEMENTATION LANGUAGE - RESOURCE

```
resources/vhost.rb

actions :create, :delete

default_action :create

attribute :site_name, String, name_attribute: true
attribute :site_port, Integer, default: 80
```

Within the resources file you specify the available actions, the default action, and the supported attributes that can be used when specifying the resource.

### 3 Light-Weight Resource-Providers (LWRP)

#### IMPLEMENTATION LANGUAGE - PROVIDER

```
providers/vhost.rb
```

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

  templates "/srv/apache/#{new_resource.site_name}/html" do
    source 'conf.erb'
    mode '0644'
    variables(document_root: "/srv/apache/#{new_resource.site_name}/html",
              port: new_resource.site_port)
    mode '0755'
    notifies :restart, 'service[httpd]'
  end

  # ... remaining resources ...
end
```

Within the provider definition you specify action blocks for each of the actions defined in the resource file. Within the action you specify resources as if you are defining a small recipe. The attributes defined for the resource are available within the action through a local variable or method named 'new\_resource'.

### 3 Light-Weight Resource-Providers (LWRP)

#### USAGE

recipes/default.rb

```
apache_vhost 'welcome' do
  action :delete
end

apache_vhost 'users'

apache_vhost 'admins' do
  site_port 8080
end
```

The name of the cookbook is combined with the name of the resource/provider file name with an underscore to create the user defined resource. This was explicitly defined in the HWRP but is automatically generated.

Otherwise this is the same results as the one defined by the HWRP.

### 3 Light-Weight Resource-Providers (LWRP)

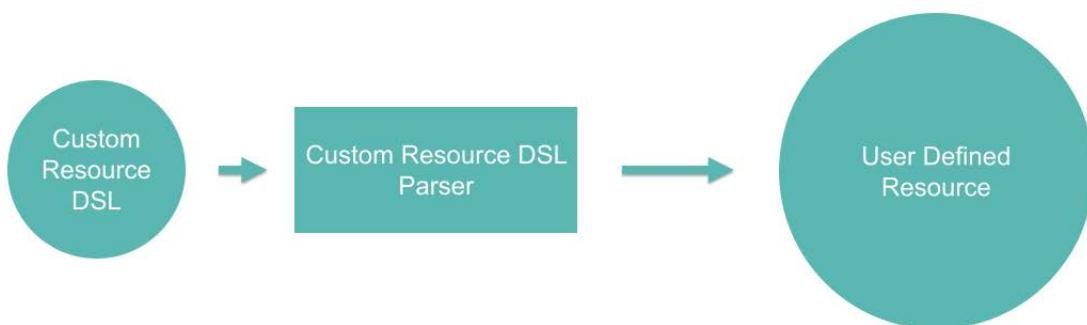
#### BENEFITS & DRAWBACKS

- Available in 0.7.12 version of Chef
- Allows for a real resource definition without understanding Ruby (vs. HWRP)
- Resource and provider implementation require learning a new DSL
- Complete resource definition is spread across two files

Implementing resources with LWRP is not the favored way to develop a resource in later versions of Chef (Chef 12.5). However, they are still in wide use within older cookbooks like those found within the Chef Supermarket.

## 4 Custom Resources

### DESCRIPTION



Custom Resources are Chef resources defined in a Domain Specific Language (DSL) that allow you to create resources without having to understand the complexity presented by HWRP. At its core it is a simplification of the work done with LWRP.

An custom resource is as much a resource as the core resources defined in Chef. A custom resource definition is defined in a single file that resides in the 'resources' directory. This file is parsed after the cookbook is synchronized and loaded. The custom resource DSL is then converted into Ruby class at runtime.

## 4 Custom Resources

### STRUCTURE

#### my\_cookbook

```
resources/
• [my_resource_name].rb
```

A custom resource is defined in a single file within the resources directory.

Within the file in the 'resources' directory you define the interface and the implementation for the custom resource. This is written in a custom resource DSL where you can specify the name of the resource, the default action, the properties that may be set, and all the actions that the resource supports.

## 4 Custom Resources

### IMPLEMENTATION LANGUAGE

```
resources/vhost.rb

resource_name :apache_vhost

property :site_name, String, name_attribute: true
property :site_port, Integer, default: 80

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

  # ... remaining resources ...
end

# ... remaining actions ...
```

The custom resource implementation is similar to the LWRP except all of the details that describe the resource are combined into a single file. The custom resource DSL is similar to one defined for the LWRP resource and LWRP provider DSL. It is an evolution of the LWRP implementation with some minor changes. The attributes are instead called properties and when used within the action implementations they no longer require the 'new\_resource' local variable or method. The default action is assumed to be the first action defined in this file: create.

## 4 Custom Resources

### USAGE

recipes/default.rb

```
apache_vhost 'welcome' do
  action :delete
end

apache_vhost 'users'

apache_vhost 'admins' do
  site_port 8080
end
```

The result is the same here as the HWRP and LWRP.

The default action is determined by the first action listed in the custom resource definition.

## 4 Custom Resources

### BENEFITS & DRAWBACKS

- Available in 12.5.0 version of Chef
- Allows for a real resource definition without understanding Ruby (vs. HWRP)
- Complete resource definition is defined in a single file (vs. LWRP)
- Custom resource implementation require learning a new DSL

Implementing resources with a custom resource is the current favored way to develop a resource for versions of Chef 12.5.X or greater. They are easier to implement than a pure Ruby implementation and are defined in a single file compared to the LWRP implementation.

# CONCEPT



## Approaches to Extending Resources

- 1 Pure Ruby (Heavy-Weight Resource-Providers / HWRP)
- 2 Definitions
- 3 Light-Weight Resource-Providers (LWRP)
- 4 Custom Resources

As you can see there are more than a few ways to extend Chef and create a resource or resource-like implementation within your recipes.

# DISCUSSION



## Discussion

Which approaches require you to define your solution in two separate files?

What are the limitations of choosing the Definitions approach?

What are some differences between LWRP and Custom Resources?

Given a Chef version prior to 12.5.0, which approach would you choose?

As a group, let's answer these questions.

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?

What questions can we answer for you?



## Why Use Custom Resources

As you can see there are more than a few ways to extend Chef and create a resource or resource-like implementation within your recipes. But before we do that, it is important to understand the value that a custom resource brings to a recipes.

# Objectives

After completing this module, you should be able to:

- Determine when a Custom Resource would be beneficial for clarity and reusability

After completing this module you will be able to describe when a Custom Resource would be beneficial for clarity and reusability.

# EXERCISE



## Evaluation Before Pursuit

*Just because I can does not mean I should. It is important to implement solutions that are arguably better software design.*

**Objective:**

- Define the judgment criteria
- Evaluate a code sample

As an group exercise we are going to look at a series of resources and discuss their quality. Quality can be rather variable unless we select a criteria for which to judge it.

# CONCEPT



## Software Quality Standards

When defining resources within our recipes we are writing software. Software has a number of quality characteristics that have already been defined. ISO/IEC 9126 is an international standard for evaluation of software quality.

When defining resources within our recipes we are writing software. Software has a number of quality characteristics that have already been defined. ISO/IEC 9126 is an international standard for evaluation of software quality.

# CONCEPT



## Software Quality Standards

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

This standard identifies 6 main quality characteristics. Let's talk about each one of these so that we have a shared understanding of what we mean when using them in this exercise.

# CONCEPT



## Software Quality Standards

- **Functionality**
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Does the code accomplish what it is designed to accomplish?

Functionality is the essential purpose of any product or service. Does the code accomplish what it is designed to accomplish? Functionality may also be concerned with if it does so securely and within compliance guidelines.

# CONCEPT



## Software Quality Standards

- Functionality
- **Reliability**
- Usability
- Efficiency
- Maintainability
- Portability

Is the solution able to withstand fault and recover from a failure?

Reliability is a judgment of whether the code accomplishes its functional goal consistently, is able to withstand fault, and recover from a failure.

# CONCEPT



## Software Quality Standards

- Functionality
- Reliability
- **Usability**
- Efficiency
- Maintainability
- Portability

Is the code easy to understand?  
Is it easy to learn?

Usability refers to the ease of use for the given code. Is the code easy to understand?  
Is it easy to learn? Does it adhere to common team standards?

# CONCEPT



## Software Quality Standards

- Functionality
- Reliability
- Usability
- **Efficiency**
- Maintainability
- Portability

Does the code consume too many physical resources when it executes (e.g. CPU, memory)?

Efficiency is concerned with the system resources required to achieve the functionality. We may consider the time, CPU, memory, network requirements, or physical space it takes to accomplish the intended operation.

# CONCEPT



## Software Quality Standards

- Functionality
- Reliability
- Usability
- Efficiency
- **Maintainability**
- Portability

Are you able to easily adapt the solution? Is it testable?

Maintainability measures the code to see if it is supportable. If there is a failure are you able to quickly identify the issue? Are you able to easily adapt the solution? Is it testable?

# CONCEPT



## Software Quality Standards

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- **Portability**

Can the software adapt to changes in its environment? Or changes to its requirements?

Portability refers to how well the software can adapt to changes in its environment or with its requirements. This may also include evaluating code for its adaptability and maybe even be easily replaced.

# EXERCISE



## Examine the Code Sample

*With the criteria defined we can now examine code samples...*

**Objective:**

- Define the judgment criteria
- Evaluate a code sample

Let's examine this first example and apply the criteria that we have defined.

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

**Functionality | Reliability | Usability | Efficiency | Maintainability | Portability**

**Does the code accomplish what it is designed to accomplish?**

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

Functionality | **Reliability** | Usability | Efficiency | Maintainability | Portability

Is the solution able to withstand fault and recover from a failure?

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

Functionality | Reliability | **Usability** | Efficiency | Maintainability | Portability

Is the code easy to understand? Is it easy to learn?

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

Functionality | Reliability | Usability | **Efficiency** | Maintainability | Portability

Does the code consume too many physical resources when it executes (e.g. CPU, memory)?

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

Functionality | Reliability | Usability | Efficiency | **Maintainability** | Portability

Are you able to easily adapt the solution? Is it testable?

## Resource Implementation v Custom Resource

```
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
```

```
apache_vhost 'admins' do
  site_port 8080
end
```

Functionality | Reliability | Usability | Efficiency | Maintainability | **Portability**

Can the software adapt to changes in its environment? Or changes to its requirements?

# EXERCISE



## Evaluation Before Pursuit

*There are many ways to critically evaluate code ... if these do not suit your or your team find the ones that do; talk about them and share them.*

**Objective:**

- ✓ Define the judgment criteria
- ✓ Evaluate a code sample

We've evaluated one code sample, let's look at a second one.

# DISCUSSION



## Discussion

What value does reviewing code for functionality, reliability, usability, efficiency, maintainability, portability bring?

# DISCUSSION



## Q&A

What questions can we answer for you?



## Creating a Custom Resource

# 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.

# EXERCISE



## Review the Cookbook

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

**Objective:**

- Checkout different branch of cookbook
- Review and execute the integration tests
- Review and execute the unit tests
- Review the default recipe

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

## Move in the apache cookbook



```
> cd ~/apache
```

## Add the Changed Files to Staging



```
> git add .
```

## Commit the Staged Changes



```
> git commit -m "Saving TDD work"
```

# Change Git Branches



```
> git checkout extending-cookbook
```

# EXERCISE



## Review the Cookbook

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

**Objective:**

- ✓ Checkout different branch of cookbook
- ❑ Review and execute the integration tests
- ❑ Review and execute the unit tests
- ❑ Review the default recipe

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

## Reviewing the Existing Integration Tests

```
~/apache/test/smoke/default/default_test.rb

describe command('curl http://localhost') do
  its(:stdout) { should match(/Welcome home/) }
end

describe command('curl http://localhost:8080') do
  its(:stdout) { should match(/Welcome admins/) }
end
```

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.

## Executing the Existing Integration Tests



```
> kitchen verify
```

```
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> (0m0.74s).
```

```
-----> Kitchen is finished. (0m7.37s)
```

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.

# EXERCISE



## Review the Cookbook

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

**Objective:**

- ✓ Checkout different branch of cookbook
- ✓ Review and execute the integration tests
- ❑ Review and execute the unit tests
- ❑ Review the default recipe

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

## Reviewing the Existing Unit Tests

```
~/apache/spec/unit/recipes/default_spec.rb

require 'spec_helper'

describe 'apache::default' do
  context 'When all attributes are default, on an CentOS 6.9' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')
      runner.converge(described_recipe)
    end
    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end
  end
end

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

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.

## Reviewing the Existing Unit Tests

```
~/apache/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.

# Reviewing the Existing Unit Tests

```
~/apache/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.

# Reviewing the Existing Unit Tests

```
~/apache/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').with_content('<h1>Welcome admins!</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.

## Reviewing the Existing Unit Tests

```
~/apache/spec/unit/recipes/default_spec.rb
```

```
# ... CONTINUES FROM THE PREVIOUS SLIDE ...
end # describe admin site
end # context
end # describe 'apache::default'
```

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.

## Executing the Existing Unit Tests



```
> chef exec rspec
```

```
.....
```

```
Finished in 0.92866 seconds (files took 2.76 seconds to load)
8 examples, 0 failures
```

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

# EXERCISE



## Review the Cookbook

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

**Objective:**

- ✓ Checkout different branch of cookbook
- ✓ Review and execute the integration tests
- ✓ Review and execute the unit tests
- ❑ Review the default recipe

Finally it is time to review the default recipe.

# Reviewing the Default Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
#  
# Copyright:: 2018, 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

```
~/apache/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.

# Reviewing the Default Recipe



~/apache/recipes/default.rb

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

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

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.

# EXERCISE



## Review the Cookbook

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

**Objective:**

- ✓ Checkout different branch of cookbook
- ✓ Review and execute the integration tests
- ✓ Review and execute the unit tests
- ✓ Review the default recipe

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

# EXERCISE



## Creating a Custom Resource

*This will make our recipe much cleaner.*

**Objective:**

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

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[/home/chef/apache/resources] action create
  - create new directory /home/chef/apache/resources
* template[/home/chef/apache/resources/vhost.rb] action create
  - create new file ~/apache/resources/vhost.rb
  - update content in file /home/chef/apache/resources/vhost.rb from none to e3b0c4
    (diff output suppressed by config)
* directory[/home/chef/apache/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'.

## Defining the Create Action

```
~/apache/resources/vhost.rb
```

```
action :create do  
  
end
```

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.

# Implementing the Create Action

```
~/apache/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.

# Refactoring the Default Recipe

```
~/apache/recipes/default.rb
```

```
#  
# Cookbook:: apache  
# Recipe:: default  
#  
# Copyright:: 2018, The Authors, All Rights Reserved.  
package 'httpd'  
  
file '/var/www/html/index.html' do  
  content '<h1>Welcome home!</h1>'  
end  
  
directory '/srv/apache/admins/html' do  
  recursive true  
  mode '0755'  
end
```

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.

# Refactoring the Default Recipe

```
~/apache/recipes/default.rb

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

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

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

## Adding the New Custom Resource

```
~/apache/recipes/default.rb

package 'httpd'

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

apache_vhost 'admins' do
  action :create
end

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

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 'apache' and the ruby file is named 'vhost' so the default name for the resource is 'apache\_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.

# Executing the Unit Tests



```
> chef exec rspec
```

```
Finished in 0.9673 seconds (files took 2.72 seconds to load)
```

```
8 examples, 3 failures
```

```
Failed examples:
```

```
rspec ./spec/unit/recipes/default_spec.rb:39 # apache::default When all attributes are default, on an unspecified platform for the admin site creates the directory  
rspec ./spec/unit/recipes/default_spec.rb:43 # apache::default When all attributes are default, on an unspecified platform for the admin site creates the configuration  
rspec ./spec/unit/recipes/default_spec.rb:47 # apache::default When all attributes are default, on an unspecified platform for the admin site creates a new home page
```

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.

# CONCEPT

## Resource Collection



Resource Collection

> rspec ➔ ●

Icon	Resource Type
<span style="color: grey;">●</span>	other resources ...
<span style="color: red;">●</span>	template '...'
<span style="color: red;">●</span>	directory '...'
<span style="color: red;">●</span>	file '...'
<span style="color: green;">●</span>	other resources ...

This resource is missing from the resource collection.

This resource is missing from the resource collection.

This resource is missing from the resource collection.

©2018 Chef Software Inc.

11-31

 CHEF

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.

# CONCEPT

A Sub-Resource Collection

**Resource Collection**

- other resources ...
- apache\_vhost
- other resources ...

step into

**Sub-Resource Collection**

- template '....'
- directory '....'
- file '....'

©2018 Chef Software Inc. 11-32 

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.

## Updating the Unit Tests to Verify Custom Resource

```
~/apache/spec/unit/recipes/default_spec.rb

require 'spec_helper'

context 'When all attributes are default, on an CentOS 6.9' do
  let(:chef_run) do
    runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9',
step_into: ['apache_vhost'])
    runner.converge(described_recipe)
  end
  it 'converges successfully' do
    expect { chef_run }.to_not raise_error
  end
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 its work.

## Executing the Unit Tests



```
> chef exec rspec
```

```
.....
```

```
Finished in 0.98788 seconds (files took 2.95 seconds to load)
8 examples, 0 failures
```

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

# Converging the Test Instance



&gt; kitchen converge

```
(up to date)
(up to date)
(up to date)
(up to date)
* service[apache] action start (up to date)

Running handlers:
Running handlers complete
Chef Client finished, 0/8 resources updated in 05 seconds
Finished converging <default-centos-67> (0m8.81s).
-----> Kitchen is finished. (0m9.80s)
```

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.

## Verifying the Test Instance



```
> kitchen verify
```

```
-----> Starting Kitchen (v1.11.1)
-----> Setting up <default-centos-67>...
      Finished setting up <default-centos-67> (0m0.00s).
-----> Verifying <default-centos-67>...
      Use `'/home/chef/apache/test/smoke/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> (0m0.70s).
-----> Kitchen is finished. (0m1.82s)
```

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

## Run a Complete Test on the Test Instance



```
> kitchen test
```

```
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> (0m0.70s).
-----> Destroying <default-centos-67>...
      ==> default: Forcing shutdown of VM...
      ==> default: Destroying VM and associated drives...
      Vagrant instance <default-centos-67> destroyed.
      Finished destroying <default-centos-67> (0m4.09s).
      Finished testing <default-centos-67> (2m21.40s).
-----> Kitchen is finished. (2m22.49s)
```

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.

# EXERCISE



## Creating a Custom Resource

*This will make our recipe much cleaner.*

**Objective:**

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

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.

# CONCEPT



## Resource Properties

A property is defined with the following syntax.

```
~/apache/resources/vhost.rb
property :site_name, String
```

The diagram illustrates the syntax of a Chef property definition:

- 1**: property method
- 2**: name (symbol)
- 2**: type
- Optional Parameters

©2018 Chef Software Inc. 11-39 

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 Integer for numbers. The optional parameters are defined as a Hash. We will explore defining a property with these parameters in the next module.

# CONCEPT



## Resource Properties

Properties are defined in the resource definition and then available within definition of the resource within the recipe.

```
~/apache/resources/vhost.rb
```

```
property :site_name, String
```

```
~/apache/recipes/default.rb
```

```
apache_vhost 'admins' do
  site_name 'admins'
  action :create
end
```

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

```
~/apache/resources/vhost.rb

property :site_name, String

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

  template "/etc/httpd/conf.d/#{new_resource.site_name}.conf" do
    source "conf.erb"
    mode "0644"
    variables(document_root: "/srv/apache/#{new_resource.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.

new\_resource.site\_name must be used to avoid warnings in Chef 13

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

# Updating the Action to use the Property

```
~/apache/resource/vhost.rb

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

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

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

## Updating the Resource to use the Property

```
~/apache/recipes/default.rb

package 'httpd'

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

apache_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 property 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'.

## Executing the Unit Tests



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 7.58 seconds to load)
8 examples, 0 failures
```

The unit tests should pass when executed.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

The integration tests should pass when executed.

# 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

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

# LAB



## apache\_vhost - site\_port Property

- Create a custom resource property named **site\_port** that is a *Integer*
- Within the apache\_vhost's create action replace the hard-coded value 8080 with the **site\_port** property
- Within the default recipe set the apache\_vhost resource named 'admins' to have a site\_port 8080

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.

Instructor Note: Allow 10 minutes to complete this exercise

## Defining a Property to Manage the site\_port

```
~/apache/resource/vhost.rb

property :site_name, String
property :site_port, Integer

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

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

## Updating the Resource to use the Property

```
~/apache/recipes/default.rb

package 'httpd'

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

apache_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 'apache\_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.

## Executing the Unit Tests



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 7.58 seconds to load)
8 examples, 0 failures
```

The unit tests should pass when executed.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

The integration tests should pass when executed.

# LAB



## apache\_vhost - site\_port Property

- ✓ Create a custom resource property named **site\_port** that is a *Integer*
- ✓ Within the apache\_vhost's create action replace the hard-coded value 8080 with the **site\_port** property
- ✓ Within the default recipe set the apache\_vhost resource named 'admins' to have a site\_port 8080

With the `site_port` property developed the 'apache\_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.

# PROBLEM



## Remove the Welcome Site

When apache installs itself it defines a default site on port 80. Up until this point we have relied on this site. We now want to remove that initial welcome site but we want to do that with a new action we will define on the custom resource we are defining.

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.

# LAB



## apache\_vhost Remove Action

- Define the 'remove' action for apache\_vhost that defines the policy:

A directory named "/srv/apache/#{\$site\_name}" 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 apache\_vhost resource named 'welcome' is removed

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.

Instructor Note: Allow 15 minutes to complete this exercise

## Defining the Resource's Remove Action

```
~/apache/resources/vhost.rb

# ... CREATE ACTION ...

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

  file "/etc/httpd/conf.d/#{new_resource.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.

## Adding the Resource with Remove Action to the Recipe

```
~/apache/recipes/default.rb

package 'httpd'

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

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

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

# ... REMAINDER OF RECIPE ...
```

When apache 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.

# LAB



## apache\_vhost Remove Action

- ✓ Define the 'remove' action for apache\_vhost that defines the policy:

**A directory named "/srv/apache/#{\$site\_name}" 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 apache\_vhost resource named 'welcome' is removed**

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

# LAB



## apache\_vhost Remove Action

- Update the default recipe's policy:

Add an apache\_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 apache\_vhost resource named 'users'
- Update the InSpec tests to expect the default site to "Welcome users!"

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.

Instructor Note: Allow 15 minutes to complete this exercise

## Adding the Resource to create the users site

```
~/apache/recipes/default.rb
```

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

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

apache_vhost 'admins' do
  site_name 'admins'
  site_port 8080
  action :create
  notifies :restart, 'service[httpd]'
end
```

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

## Removing the Un-needed Unit Test Expectation

```
~/apache/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 ...
```

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

## Adding Expectations for the users Site

```
~/apache/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
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.

## Executing the Unit Test Suite



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 2.56 seconds to load)
10 examples, 0 failures
```

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

## Updating the Expectation for the users Site

```
~/apache/test/smoke/default/default_test.rb

describe command('curl http://localhost') do
  its(:stdout) { should match(/Welcome users/) }
end

describe command('curl http://localhost:8080') do
  its(:stdout) { should match(/Welcome admins/) }
end
```

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

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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.

# LAB



## apache\_vhost Remove Action

- ✓ Update the default recipe's policy:  
Add an apache\_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 apache\_vhost resource named 'users'
- ✓ Update the InSpec tests to expect the default site to "Welcome users!"

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.

# DISCUSSION



## Discussion

What are the benefits of using a custom resource to manage the virtual hosts? What are the drawbacks of using a custom resource?

What does the resource collection look like when using a custom resource?

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.

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?

What questions can we answer for you?



## Refining a Custom Resource

# Objectives

After completing this module, you should be able to:

- Set a custom resource's name to a property
- Set a default value for a custom resource property
- Define notifications correctly when creating custom resources

After completing this module you should be able to set a custom resource's name to a property, set a default value if the property is not provided, and define notifications correctly within the custom resource.

# EXERCISE



## Refactoring the Resources

**Objective:**

- Define a default action for the custom resource
- Set the resource name as the site\_name property
- Set the default value of the site\_port property
- Move the resource notifications to the recipe

Every resource that you have used within Chef has had a default action.

# PROBLEM



## Selecting a Default Action

Resources often have a default action. The default action is often the action that is the least surprising. For most resources it is an additive/restorative action that moves the system into the desired state.

Our custom resource should be no different. Having a default action that performs a non-surprising operation is important. Of the two actions that we have defined the create action seems like the current correct default action.

# CONCEPT



## The First Action is the Default

The first action within the custom resource definition is the default one. But you can explicitly define the default action within the resource definition.

If you defined the `create` action as the first action within a resource definition file it automatically is the default action. The first action is the default action and that probably makes sense to those reading the custom resource definition.

You may also explicitly declare the default action.

## Defining a Default Action for the Resource

```
~/apache/resources/vhost.rb

property :site_name, String
property :site_port, Fixnum

default_action :create

action :create do
  directory "/srv/apache/#{new_resource.site_name}/html" do
    recursive true
    mode "0755"
  end
# ... REMAINING RESOURCE DEFINITION ...
```

If the first action is the create action then you do not need to explicitly define the default action. However, you may decide that it makes it clearer for you or those you are collaborating with to specify this within the resource definition.

## Removing the Default Action for Resources

```
~/apache/recipes/default.rb

# ... INITIAL RECIPE DEFINITION ...

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

apache_vhost 'admins' do
  site_port 8080
  site_name 'admins'
  action :create
end
```

With the default action defined all uses of the custom resource which explicitly defined it may have those lines removed.

## Executing the Unit Test Suite



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 2.56 seconds to load)
10 examples, 0 failures
```

All the unit tests should pass.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

All the integration test should pass.

# EXERCISE



## Refactoring the Resources

**Objective:**

- ✓ Define a default action for the custom resource
- Set the resource name as the site\_name property
- Set the default value of the site\_port property
- Move the resource notifications to the recipe

Now we will look at tying the name provided to the custom resource to the site\_name property.

# PROBLEM



## Clarity in the Custom Resource

There is some duplication in the declaration of the resource. The name of the resource and the site\_name. We want to default to use the name specified in the resource as the site\_name. This is similar to other resources.

We want to ensure our custom resource is clear and concise. At the moment when you define the resource within the recipe you specify a value as the name of the custom resource and then a property that matches that same name. This seems like a redundancy that we want to remove.

## Tying the Resource Name to the Property

```
~/apache/resources/vhost.rb

property :site_name, String, name_attribute: true
property :site_port, Fixnum

default_action :create

action :create do
  directory "/srv/apache/#{new_resource.site_name}/html" do
    recursive true
    mode "0755"
  end
# ... REMAINING RESOURCE DEFINITION ...
```

One property may have the 'name\_attribute' option set to true. This property will be automatically populated from the name of the resource. Using the name of the resource will allow us to remove the need to specify that property which is repeating a value within the use of the custom resource.

## Removing the site\_name Property

```
~/apache/recipes/default.rb

package 'apache'

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

apache_vhost 'admins' do
  site_name 'users'
  site_port 80
end
```

Now that the site\_name property is set as the name attribute we can remove the site name property from each use of the custom resource.

## Removing the site\_name Property

~/apache/recipes/default.rb

```
# ... INITIAL RECIPE DEFINITION ...

apache_vhost 'admins' do
  site_name 'admins'
  site_port 8080
end

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

Now that the site\_name property is set as the name attribute we can remove the site name property from each use of the custom resource.

## Executing the Unit Test Suite



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 2.56 seconds to load)
10 examples, 0 failures
```

All the unit tests should pass.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

All the integration test should pass.

# EXERCISE



## Refactoring the Resources

**Objective:**

- ✓ Define a default action for the custom resource
- ✓ Set the resource name as the site\_name property
- Set the default value of the site\_port property
- Move the resource notifications to the recipe

Resource properties can also have default values setup for them. Lets explore setting a default value for the site\_port property.

# CONCEPT



## Setting Default Values for Properties

Properties can also have default values. When deploying a website the default port that one would expect to see that site is on port 80.

Setting a default value for a property allows you to define the resource and if you omit setting the property then the default value is used. We can use this behavior for our `site_port`, choosing to say that if you do not specify a port we want the port to be 80.

## Setting a Default Value for the Property

```
~/apache/resources/vhost.rb

property :site_name, String, name_attribute: true
property :site_port, Fixnum, default: 80

default_action :create

action :create do
  directory "/srv/apache/#{new_resource.site_name}/html" do
    recursive true
    mode "0755"
  end
# ... REMAINING RESOURCE DEFINITION ...
```

All properties may have a default value. To add one requires that you simply add the option 'default' with its corresponding default value. Here we are setting the 'site\_port' value to 80.

## Removing the site\_port Property

```
~/apache/recipes/default.rb

package 'apache'

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

apache_vhost 'users' do
  site_port 80
end
```

This allows us to remove the value from a single resource.

## Executing the Unit Test Suite



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 2.56 seconds to load)
10 examples, 0 failures
```

All the unit tests should pass.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

All the integration test should pass.

# EXERCISE



## Refactoring the Resources

**Objective:**

- ✓ Define a default action for the custom resource
- ✓ Set the resource name as the site\_name property
- ✓ Set the default value of the site\_port property
- ❑ Move the resource notifications to the recipe

Finally we want to properly address how the custom resource handles notifications.

# PROBLEM



## Resource Notifications

Defining a notification in a resource within a custom resource creates a fragile relationship. One that we want to address by removing any notifications to outside resources.

When defining notifications within a resource action if you reference a resource outside of the action implementation there is a chance that your code may break if that resource's name were to change or simply not be implemented at all.

If any resource were to take action within the custom resource then the custom resource considers itself as taking action. We often say that any changed resource events are sent the parent custom resource and from that custom resource you can define your notifications.

## Removing the Notification from the Resource

```
~/apache/resources/vhost.rb

# ... INITIAL RESOURCE DEFINITION ...

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

First we remove the dependency on a resource not present within the action implementation of the custom resource.

## Adding the Notification to the Resources

```
~/apache/recipes/default.rb

package 'apache'

apache_vhost 'welcome' do
  notifies :restart, 'service[httpd]'
  action :remove
end

apache_vhost 'users' do
  notifies :restart, 'service[httpd]'
end
```

We then add the notifications to the custom resource implementations within the default recipe.

## Adding the Notification to the Resources

```
~/apache/recipes/default.rb

# ... INITIAL RECIPE DEFINITION ...

apache_vhost 'admins' do
  site_port 8080
  notifies :restart, 'service[httpd]'
end

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

We then add the notifications to the custom resource implementations within the default recipe.

## Executing the Unit Test Suite



```
> chef exec rspec
```

```
.....
```

```
Finished in 1.22 seconds (files took 2.56 seconds to load)
10 examples, 0 failures
```

All the unit tests should pass.

## Converging and Verifying the Test Instance



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

```
-----> Verifying <default-centos-67>...
      Use `/home/chef/apache/test/smoke/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)
```

All the integration test should pass.

# EXERCISE



## Refactoring the Resources

**Objective:**

- ✓ Define a default action for the custom resource
- ✓ Set the resource name as the site\_name property
- ✓ Set the default value of the site\_port property
- ✓ Move the resource notifications to the recipe

We now have refactored the custom resource to have it behave more like other resources we are familiar within Chef.

# DISCUSSION

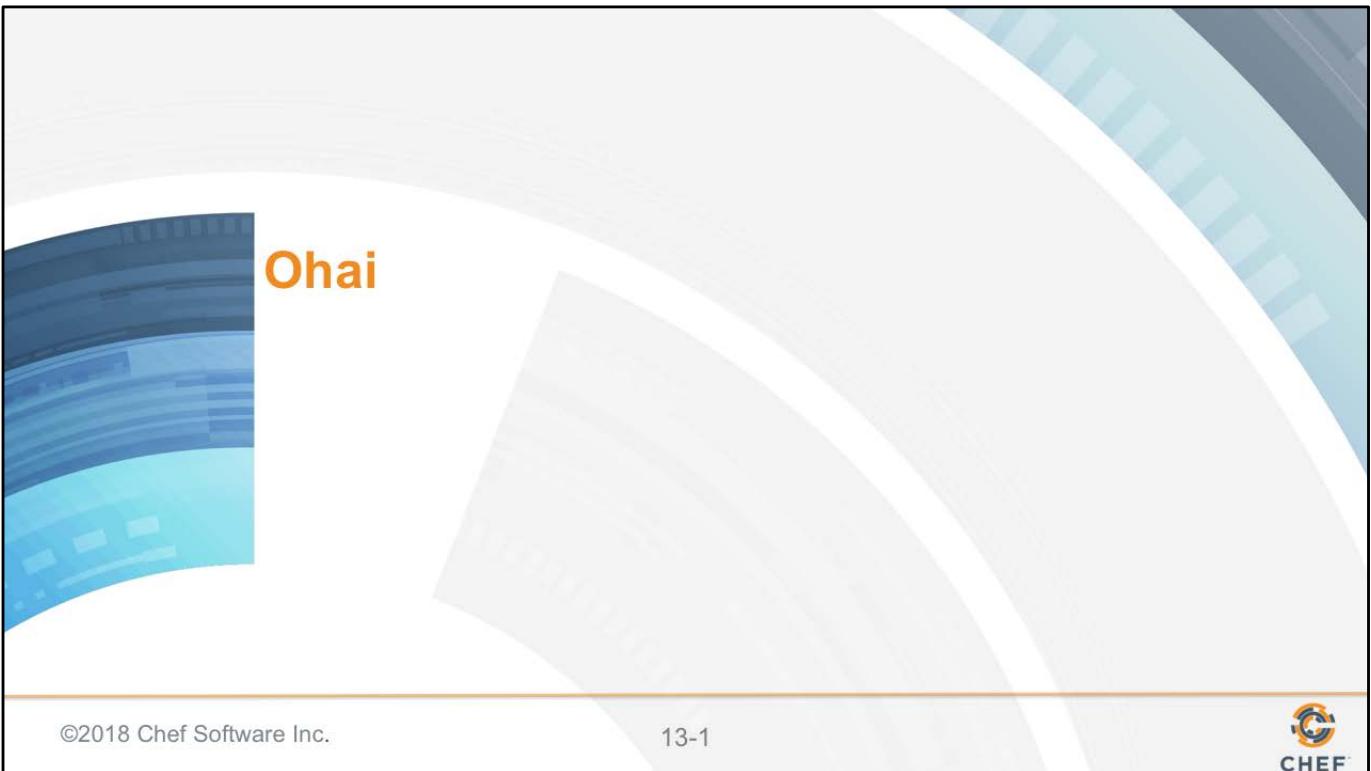


## Q&A

What questions can we answer for you?

What questions can we answer for you?





## Ohai

©2018 Chef Software Inc.

13-1



Before you set out to start managing your nodes it is important to understand the current state of your nodes. As Chef, a platform agnostic tool, is written in Ruby, a platform agnostic language, it is useful to understand what is or is not installed on the system. This information is helpful in helping a resource select the correct provider or for that provider to determine which version of the tool or language is at its disposal.

# Objectives

After completing this module, you should be able to:

- Execute the Ohai command-line tool to return an attribute
- Describe when Ohai is loaded in the chef-client run
- Describe when new attributes for the node are stored
- Describe precedence of attributes collected by Ohai

After completing this module you will be able to execute the Ohai command-line tool to return an attribute, describe when Ohai is loaded in the chef-client run, when new attributes for the node are stored in that chef-client run, and be able to describe the attribute precedence for attributes collect by Ohai.

# CONCEPT



## Ohai

Ohai is a tool that is used to detect attributes on a node, and then provide these attributes to the chef-client at the start of every chef-client run. The types of attributes Ohai collects include (but are not limited to):

- Platform details
- Network usage
- Memory usage
- CPU data

Ohai is a tool that is used to detect attributes on a node, and then provide these attributes to the chef-client at the start of every chef-client run. Ohai is required by the chef-client and must be present on a node. (Ohai is installed on a node as part of the chef-client install process.)

# EXERCISE



## Exploring Ohai

*To understand Ohai we must explore it in isolation, understand where it fits in the ecosystem, and how the data it provides is stored.*

**Objective:**

- Execute Ohai to retrieve details about the node
- View Ohai's execution within a chef-client run
- Describe attributes precedence

As a group we will explore using Ohai from the command-line then view how it is executed within a chef-client run and then talk about the attributes that it collects. We'll start with ohai the command-line tool.

# CONCEPT



## All About The System

Ohai queries the operating system with a number of commands, similar to the ones demonstrated.

The data is presented in JSON (JavaScript Object Notation).

Ohai, the command-line application, will output all the system details represented in JavaScript Object Notation (JSON).

## Running Ohai to Show All Attributes



&gt; ohai

```
{  
  "kernel": {  
    "name": "Linux",  
    "release": "2.6.32-431.1.2.0.1.el6.x86_64",  
    "version": "#1 SMP Fri Dec 13 13:06:13 UTC 2013",  
    "machine": "x86_64",  
    "os": "GNU/Linux",  
    "modules": {  
      "veth": {  
        "size": "5040",  
        "refcount": "0"  
      },  
      "ipt_addrtype": {  
        "size": "5040",  
        "refcount": "0"  
      }  
    }  
  }  
}
```

Ohai is also a command-line application that is part of the ChefDK. When you run it you will see the entire JSON representation of the system.

## Running Ohai to Show the IP Address



```
> ohai ipaddress
```

```
[
```

```
"172.31.57.153"
```

```
]
```

You can also run ohai with a parameter. In this case when we want only the ipaddress from the entire body of information we can provide it as a parameter.

## Running Ohai to Show the Hostname



```
> ohai hostname
```

```
[
```

```
"ip-172-31-57-153"
```

```
]
```

Similar, we can specify the hostname to return only the hostname of the system.

## Running Ohai to Show the Memory



```
> ohai memory
```

```
{
  "swap": {
    "cached": "0kB",
    "total": "0kB",
    "free": "0kB"
  },
  "hugepages": {
    "total": "0",
    "free": "0",
    "reserved": "0",
    "surplus": "0"
  },
  "total": "1018184kB",
  "free": "280972kB",
  "buffers": "54340kB",
}
```

When we ask for the memory of the system we receive a hash that contains a number of keys and values.

## Running Ohai to Show the Total Memory



```
> ohai memory/total
```

```
[
```

```
"1018184kB"
```

```
]
```

We can grab a single value, like the total memory, by specifying a slash between the top-level key and the next level key underneath it. This command will return the total memory of the system.

## Running Ohai to Show the CPU



```
> ohai cpu
```

```
{  
  "0": {  
    "vendor_id": "GenuineIntel",  
    "family": "6",  
    "model": "63",  
    "model_name": "Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz",  
    "stepping": "2",  
    "mhz": "2400.078",  
    "cache_size": "30720 KB",  
    "physical_id": "0",
```

We can return all the details about the cpu. We see that there is one cpu, named '0', that contains more information.

## Running Ohai to Show the First CPU



```
> ohai cpu/0
```

```
{  
  "vendor_id": "GenuineIntel",  
  "family": "6",  
  "model": "63",  
  "model_name": "Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz",  
  "stepping": "2",  
  "mhz": "2400.078",  
  "cache_size": "30720 KB",  
  "physical_id": "0",  
  "core_id": "0",
```

Here we are asking for all the details about the cpu named '0'.

## Running Ohai to Show the First CPU Mhz



```
> ohai cpu/0/mhz
```

```
[
```

```
"2400.078"
```

```
]
```

And finally if we wanted to display the Megahertz of that specific cpu we can append an additional key to the parameter.

# CONCEPT



## Ohai is Composed of Plugins

Ohai is packaged with a core set of plugins that are automatically loaded when executing Ohai.

These plugins provide the attributes we see in the JSON output (e.g. ipaddress, hostname, memory, cpu).

Ohai

Example Plugins

### NetworkAddresses

ipaddress, ip6address, macaddress

### Hostname

hostname, domain, fqdn, machinename

### Memory

memory, memory/swap

### CPU

cpu

Ohai is composed of plugins that collect these different attributes. When you execute Ohai it will load the core plugins that are packaged with it.

# CONCEPT



## Custom Ohai Plugins

It is possible to define your own plugins and have Ohai load those plugins.

```
> ohai -d PATH_TO_CUSTOM_PLUGINS
```

We will explore creating an Ohai plugin and loading it with Ohai from the command-line in the 'Creating Ohai Plugins' module.

Ohai is composed of plugins that collect these different attributes. When you execute Ohai it will load the core plugins that are packaged with it.

# EXERCISE



## Exploring Ohai

*To understand Ohai we must explore it in isolation, understand where it fits in the ecosystem, and how the data it provides is stored.*

**Objective:**

- ✓ Execute Ohai to retrieve details about the node
- ❑ View Ohai's execution within a chef-client run
- ❑ Describe attributes precedence

Executing ohai from the terminal gives you an idea about all the data that Ohai can provide for a system. Now it is important to see where this data is captured in the chef-client run.

# CONCEPT

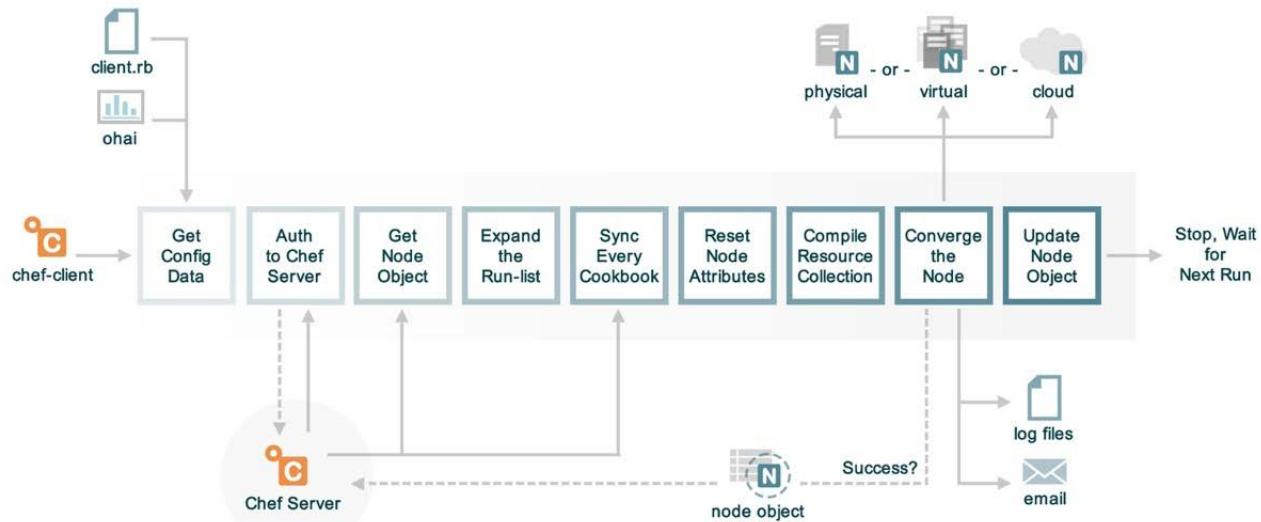


## chef-client

chef-client automatically executes ohai and stores the data about the node in an object we can use within the recipes. When the chef-client run completes successfully the details about the node are sent to the Chef Server.

<http://docs.chef.io/ohai.html>

## The Anatomy of a chef-client Run



## Running Ohai in Code



```
> chef exec pry
```

```
[1] pry(main)> require 'ohai'  
=> true  
[2] pry(main)> ohai = Ohai::System.new  
=> #<Ohai::System:0x007fc62fadcc490 @plugin_pat...@safe_run=true>>  
[3] pry(main)> ohai.all_plugins('ipaddress')  
[4] pry(main)> ohai.all_plugins('hostname')  
[5] pry(main)> ohai.all_plugins('memory')  
[6] pry(main)> ohai.all_plugins('cpu')  
[7] pry(main)> ohai.all_plugins  
[8] pry(main)> exit
```

chef-client run ohai in code as one of it's first steps. We can examine how that is done with Pry. Pry can be used as a debugger and as a REPL (Read-Evaluate-Print-Loop) tool. We can run this to allow to explore how Ohai is executed by the chef-client application.

First launch the session by running the specified command. Within this interactive session you can load the Ohai gem with the require command, create a new Ohai System object, and then ask the ohai object to load specific plugins or all plugins through the 'all\_plugins' method. When you are done you can exit by entering the command 'exit'.

Type 'q' to exit large outputs

# EXERCISE



## Exploring Ohai

*To understand Ohai we must explore it in isolation, understand where it fits in the ecosystem, and how the data it provides is stored.*

**Objective:**

- ✓ Execute Ohai to retrieve details about the node
- ✓ View Ohai's execution within a chef-client run
- ❑ Describe attributes precedence

chef-client loads and executes Ohai within Ruby. Ohai returns Ruby object representations of the data that chef-client is able to evaluate and store within the node object. These attributes discovered by Ohai become attributes of the node object and it is important to take a quick moment to discuss how these attributes compare to the other attributes that may be defined in other locations within cookbooks, roles, and environments.

# CONCEPT



## Node Attributes

A node object maintains the attributes collected from Ohai, from the previous node object (as returned by the Chef Server), from the environments, roles, and the cookbooks defined in the run list.

Later within the chef-client a node object is created with the attributes collected from Ohai, the values previously stored on the Chef Server, and then the attributes defined in the environments, roles and cookbooks described in the node's run list. The node prioritizes and gives precedence to the attributes collected by Ohai.

## Viewing Attribute Precedence as a Table

LOCATION	Attribute Files	Node / Recipe	Environment	Role	
LEVEL	default	1	2	3	4
	force_default	5	6		
	normal	7	8		
	override	9	10	12	11
	force_override	13	14		
	automatic		15		
			Ohai		

This is a table representation of the various levels of precedence that can be specified with the location in which it can be specified. The lower the value, the lower the precedence. The higher the value, the higher the precedence.

The attributes collected from Ohai are considered automatic attributes granting them the value of 15. This means all data collected through Ohai attributes cannot ever be overridden.

That should make sense based on the data that we have queried so far in this module (e.g. CPU, memory). Never would we want to have an attribute defined in a cookbook or environment override this data collected about our system. This is also important when considering whether you want to create an Ohai plugin. The kind of data that you want to collect should not be data that you will want to override as it is data that describes the system and not data that you want to configure the system.

# EXERCISE



## Exploring Ohai

*To understand Ohai we must explore it in isolation, understand where it fits in the ecosystem, and how the data it provides is stored.*

**Objective:**

- ✓ Execute Ohai to retrieve details about the node
- ✓ View Ohai's execution within a chef-client run
- ✓ Describe attributes precedence

We have seen how to use Ohai as a command-line tool, explored how chef-client uses it, and seen the precedence level at which this data is stored. In the next module we will discuss Ohai's plugin history, its plugin structure, and the DSL (Domain Specific Language) it provides to express these plugins.

# DISCUSSION



## Discussion

When might you execute ohai from the command-line to gather data about the system?

Why might it be important to collect details about the system, through Ohai, early in the chef-client run?

What kind of data should be collected and stored within Ohai? What kind of data should it not collect?

Let's finish with a discussion.

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?

What questions can we answer for you?



## Ohai Plugins

# CONCEPT



## Ohai is Composed of Plugins

Ohai is packaged with a core set of plugins that are automatically loaded when executing Ohai.

These plugins provide the attributes we see in the JSON output (e.g. ipaddress, hostname, memory, cpu).

Ohai

Example Plugins

### NetworkAddresses

ipaddress, ip6address, macaddress

### Hostname

hostname, domain, fqdn, machinename

### Memory

memory, memory/swap

### CPU

cpu

As we saw in the previous module Ohai provides a large set of attributes that it provides through plugins. All the data that Ohai collects are stored in plugins. Ohai comes packaged with a core set of plugins that capture a lot of common data across many different platforms.

# Objectives

After completing this module, you should be able to:

- Find Ohai's core plugins
- Express what a plugin provides, depends on, and how it collects its data

After completing this module you will be able to find the plugins that come packaged core with Chef, express what a plugin provides, depends on, and how it collects its data

# EXERCISE



## Reviewing the Ohai Gem

*Ohai is a Rubygem. First we need to learn about how a gem is structured.*

### **Objective:**

- Review the basic structure of the Ohai gem
- Review the 'language' plugin
- Review the 'python' plugin

To review the core plugins packaged with Ohai we need to spend some time reviewing the source code of the gem as none of the gems are defined in documentation.

# CONCEPT



## Ohai is Ruby Gem

Ruby gems are the ways in which Ruby developers share the code that they develop with others. A Ruby gem is really a packaging structure similar to that of a Chef cookbook.

Ohai is a Ruby gem that is packed in the Chef Development Kit (Chef DK). A Ruby gem is a packaging structure that allows for the code to be reused and shared.

# Searching for a Gem on Rubygems

## Steps

1. Visit <https://rubygems.org>
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.

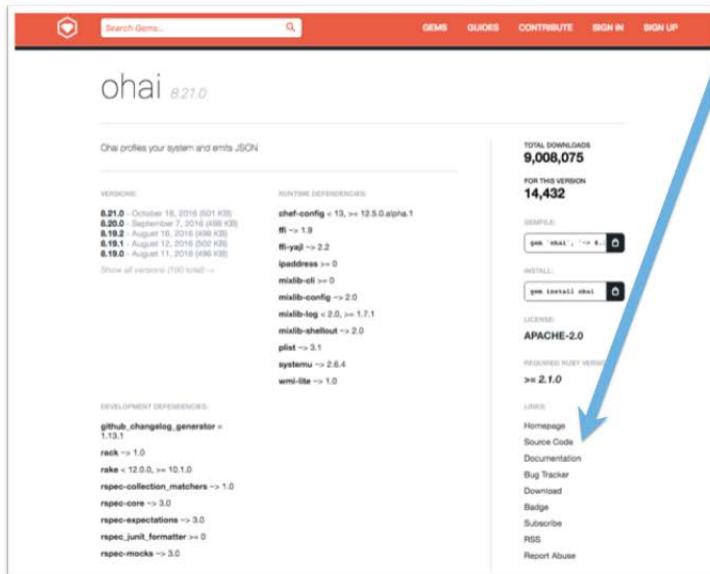


All Rubygems are stored on rubygems.org. We can come to the site and search for any gem by their name. Search for the Rubygem named "ohai".

# Searching for a Gem on Rubygems

## Steps

1. Visit <https://rubygems.org>
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.



©2018 Chef Software Inc.

14-7

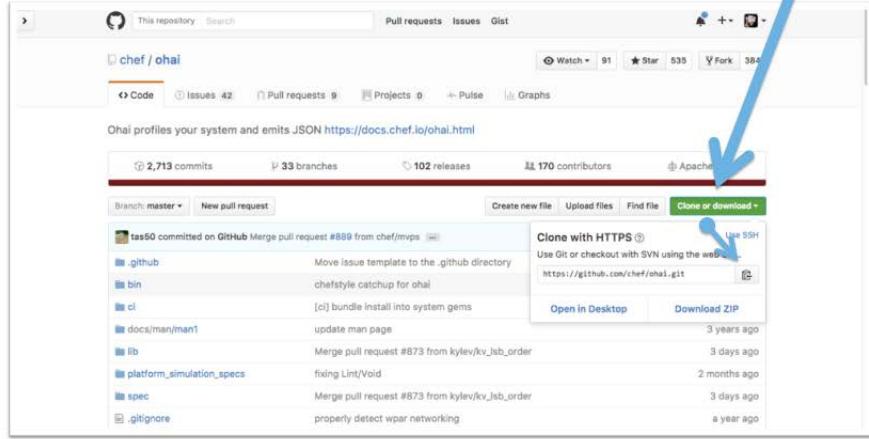


The project page for the gem itself contains important information about the releases, where to find the source, where to file issues, etc. We are interested in viewing the source of the project so we want to click on that link.

# Searching for a Gem on Rubygems

## Steps

1. Visit <https://rubygems.org>
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.



The ohai project is stored as a git repository within the Chef organization on GitHub. We can clone this project to our workstation to give us the ability to review the source code.

## Returning to the Home Directory



```
> cd ~
```

We are going to obtain the Ohai library on our local workstation so let's start by returning to the home directory.

## Cloning the Ohai Library



```
> git clone https://github.com/chef/ohai.git
```

```
Cloning into 'ohai'...
remote: Counting objects: 20571, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 20571 (delta 7), reused 0 (delta 0), pack-reused 20540
Receiving objects: 100% (20571/20571), 4.46 MiB | 2.13 MiB/s, done.
Resolving deltas: 100% (13408/13408), done.
Checking connectivity... done.
```

Git is installed with the Chef DK so we will use it to clone the Ohai project.

## Viewing the Contents of the Project



```
> tree ohai
```

```
└── ohai
    ├── appveyor.yml
    ├── bin
    │   └── ohai
    ├── CHANGELOG.md
    ├── ci
    │   ├── jenkins_run_tests.bat
    │   └── jenkins_run_tests.sh
    ├── docs
    │   └── man
    │       └── man1
    │           └── ohai.1
```

The gem contains several important items within the top-level directory. We are going to explore the contents of some of the essential files.

# Viewing the README

```
~/ohai/README.md
```

```
# ohai

[![Build Status Master] (https://travis-ci.org/chef/ohai.svg?branch=master) (https://travis-ci.org/chef/ohai) [![Build Status
Master] (https://ci.appveyor.com/api/projects/status/github/chef/ohai?branch=master&svg=true&
assingText=master%20-%200k&pendingText=master%20-%20Pending&failingText=master%20-
%20Failing)] (https://ci.appveyor.com/project/Chef/ohai/branch/master) [![Gem
Version] (https://badge.fury.io/rb/ohai.svg)] (https://badge.fury.io/rb/ohai)
```

```
## Description
```

Ohai detects data about your operating system. It can be used standalone, but its primary purpose is to provide node data to Chef.

Ohai will print out a JSON data blob for all the known data about your system. When used with Chef, that data is reported back via node attributes.

Chef distributes ohai as a RubyGem. This README is for developers who want to modify the Ohai source code. For users who want to write plugins for Ohai, see the docs:

The README contains information on how to install, configure and use this gem. This is often the place to start when exploring the gem.

# Viewing the Gem Specification

```
~/ohai/ohai.gemspec

$:.unshift File.expand_path("../lib", __FILE__)
require "ohai/version"

Gem::Specification.new do |s|
  s.name = "ohai"
  s.version = Ohai::VERSION
  s.platform = Gem::Platform::RUBY
  s.summary = "Ohai profiles your system and emits JSON"
  s.description = s.summary
  s.license = "Apache-2.0"
  s.author = "Adam Jacob"
  s.email = "adam@chef.io"
  s.homepage = "https://docs.chef.io/ohai.html"
```

The gem specification defines important information about the Rubygem. Within it you will find metadata that describes the owner, licensing, contact information, dependencies, development dependencies, the files to package in the gem, and which one of those are executables.

## Viewing the lib Directory



```
> tree ohai/lib
```

```
ohai/lib
├── ohai
│   ├── application.rb
│   ├── common
│   │   └── dmi.rb
│   ├── config.rb
│   ...
│   └── version.rb
└── ohai.rb
19 directories, 161 files
```

The lib (or library) directory contains the source code for this gem. Within the root of the directory you will find a single file that shares the same name as the gem.

In the previous module when we typed "require 'ohai'" this was the file that was loaded into memory.

## Viewing the ohai.rb file in the lib Directory

```
~/ohai/lib/ohai.rb
```

```
#  
# http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.  
#  
require "ohai/version"  
require "ohai/config"  
require "ohai/system"  
require "ohai/exception"
```

This file requires more files from within the gem. The paths specified are relative to the 'lib' directory so all of these examples are loading files from within the subdirectory of ohai.

## Viewing the plugins directory



```
> tree ohai/lib/ohai/plugins
```

```
ohai/lib/ohai/plugins
├── aix
│   ├── cpu.rb
│   ├── filesystem.rb
│   ├── kernel.rb
│   ├── memory.rb
│   ├── network.rb
│   ├── os.rb
│   ├── platform.rb
│   ├── uptime.rb
│   └── virtualization.rb
├── azure.rb
└── bsd
    └── filesystem.rb
```

Ohai stores its plugins in a specific subdirectory of this project.

# EXERCISE



## Reviewing the Ohai Gem

*Let's take a look at the structure of a plugin.*

**Objective:**

- ✓ Review the basic structure of the Ohai gem
- ❑ Review the 'language' plugin
- ❑ Review the 'python' plugin

That was a quick introduction to the gem structure to give us an idea about where the plugins are stored. Now it is time to explore the Domain Specific Language (DSL) used to write these plugins.

# CONCEPT



## Recent Major Ohai Releases

### Ohai 6

Released: April 13, 2011

Chef Version: 0.10.0

[docs.chef.io/release/ohai-6](https://docs.chef.io/release/ohai-6)

### Ohai 7

Released: April 8, 2014

Chef Version: 11.12.0

[docs.chef.io/release/ohai-7](https://docs.chef.io/release/ohai-7)

### Ohai 8

Released: Dec. 4, 2014

Chef Version: 11.18.0

[docs.chef.io/release/ohai-8](https://docs.chef.io/release/ohai-8)

Ohai has seen many notable releases. Depending on the version of Chef you are using within your organization may dictate which version of Ohai is being used.

# CONCEPT



## Focus on Ohai 7

Ohai 7 refined the Domain Specific Language (DSL) created in the previous version of Ohai. Ohai 8 continues to use the same language.



Ohai 6 introduced the ability to express plugins through a DSL. Ohai 7 refined that DSL. Ohai 8 continues to use that same language. The following slides and our exercise in the next module will focus on the DSL defined in Ohai 7.

# Viewing the Languages Plugin



~/ohai/lib/ohai/plugins/languages.rb

```
Ohai.plugin(:Languages) do
  provides "languages"
  collect_data do
    languages Mash.new
  end
end
```

Let's load the languages plugin and review the basic structure of the plugin.

# Viewing the Languages Plugin



~/ohai/lib/ohai/plugins/languages.rb

```
Ohai.plugin(:Languages) do
  provides "languages"
  collect_data do
    languages Mash.new
  end
end
```

## Plugin Name

- Ruby Symbol
- First Letter Capitalized

## Node attributes provided by the plugin

Code executed on all platforms and stored in the provided attribute(s).

A plugin starts with invoking a method on the Ohai class with a single parameter. That parameter provided is the symbol name of the plugin. All Ohai plugins must have a symbol name with the first letter capitalized.

The remainder of the plugin is defined within the block of the 'plugin' method. The 'provides' method specifies what attribute or attributes the plugin will be added to the node object. The 'collect\_data' method defines a block which contains the code that is executed on all platforms. This block of code will often times set the values of the attributes the plugin provides.

# Viewing the Language Plugin



~/ohai/lib/ohai/plugins/languages.rb

```
Ohai.plugin(:Languages) do
  provides "languages"
  collect_data do
    languages Mash.new
  end
end
```

The [Languages](#) plugin provides the node attribute '[languages](#)' which is populated, on all platforms, with a [Mash](#).

This plugin is named Languages. It provides the languages attribute on the node. This languages attribute is populated with the contents of a new Mash.

But what is a Mash?

# CONCEPT



## Hash

### Using a String as a key

```
[1] pry(main)> content = Hash.new  
=> {}  
[2] pry(main)> content['name'] =  
'Chef'  
=> "Chef"  
[3] pry(main)> content['name']  
=> "Chef"  
[4] pry(main)> content[:name]  
=> nil
```

### Using a Symbol as a key

```
[1] pry(main)> content = {}  
=> {}  
[2] pry(main)> content[:name] =  
'Chef'  
=> "Chef"  
[3] pry(main)> content[:name]  
=> "Chef"  
[4] pry(main)> content['name']  
=> nil
```

To understand what a Mash is first let's talk about Ruby's Hash. Hashes allow you to store values with a key; often times these keys are Ruby Strings or Ruby Symbols. When you want to retrieve that value you need to provide the same key. So if say you stored data with a Symbol key it is only retrievable with a Symbol key. The same could be said for using a String key.

# CONCEPT



## Mash

### Using a String as a key

```
[1] pry(main)> require 'chef'  
=> true  
[2] pry(main)> content = Mash.new  
=> {}  
[3] pry(main)> content['name'] = 'Chef'  
=> "Chef"  
[4] pry(main)> content['name']  
=> "Chef"  
[5] pry(main)> content[:name]  
=> "Chef"
```

### Using a Symbol as a key

```
[1] pry(main)> require 'chef'  
=> true  
[2] pry(main)> content = Mash.new  
=> {}  
[3] pry(main)> content[:name] = 'Chef'  
=> "Chef"  
[4] pry(main)> content[:name]  
=> "Chef"  
[5] pry(main)> content['name']  
=> "Chef"
```

A Mash is similar to a Ruby Hash except that it is indifferent to whether you provide it a String key or Symbol key. Either of those types of keys will return value stored by the other. This more lenient data structure allows for these two keys to be used interchangeably. Allowing us to use whichever key style we prefer without being penalized if we were to guess the key style that differs from other plugins.

# EXERCISE



## Reviewing the Ohai Gem

*Now it is time to look at a more complex plugin.*

**Objective:**

- ✓ Review the basic structure of the Ohai gem
- ✓ Review the 'language' plugin
- ❑ Review the 'python' plugin

The language plugin is small plugin that setups up a data structure for other language plugins to add more information to it. Let's review a specific language plugin to see a more complex implementation.

## Viewing the Python plugin

```
~/ohai/lib/ohai/plugins/python.rb
```

```
Ohai.plugin(:Python) do
  provides "languages/python"

  depends "languages"

  collect_data do
    begin
      so = shell_out("python -c \"import sys; print (sys.version)\"")
      # Sample output:
      # 2.7.11 (default, Dec 26 2015, 17:47:53)
      # [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
      if so.exitstatus == 0
        python = Mash.new
        output = so.stdout.split
        python[:version] = output[0]
        if output.length >= 6
          python[:release] = output[1]
          python[:patch] = output[2]
          python[:minor] = output[3]
          python[:major] = output[4]
        end
      end
    rescue
      # If we can't run python, just return nil
    end
  end
end
```

Node attributes provided by the plugin

This plugin depends on the attributes in the Languages to be defined

Here within the Python plugin we see the same structure with a dependency and a significant amount of work being done in the 'collect\_data' method block. The attribute provided by this plugin can be found on the node object under the specified path. Remember this is the same path structure you use on the command-line when wanting to traverse the attributes provided.

The dependency described here states that this plugin requires that the node attribute value 'languages' must be defined first before this plugin will execute. Ohai will determine how to execute the plugins based on these dependencies.

# Viewing the Python plugin

```
~/ohai/lib/ohai/plugins/python.rb

Ohai.plugin(:Python) do
  provides "languages/python"

  depends "languages"

  collect_data do
    begin
      so = shell_out("python -c \"import sys; print (sys.version)\"")
      # Sample output:
      # 2.7.11 (default, Dec 26 2015, 17:47:53)
      # [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
      if so.exitstatus == 0
        python = Mash.new
        output = so.stdout.split
        python[:version] = output[0]
        if output.length >= 6
```

Within the `collect_data` block we use a helper method named '`shell_out`'. This '`shell_out`' method accepts a single parameter which is the command to run. This '`shell_out`' method will generate an object for which you can ask for the standard output, standard error, and the exit status.

This command is executed and if the status is successful (0 status code) then look at the standard output, split it into multiple lines, extract the version and possibly any build date information, and then store that information into the `Mash` that was created by the `Languages` plugin. If a failure occurs at any point catch that error and display a debug message.

You will find that most Ohai plugins will fit the following pattern. Perform a system related call to collect some data, use Ruby to process that data, and then store the data.

# CONCEPT



## collect\_data for a specific Platform

Plugins can collect data in different ways across different platforms. When defining a collect\_data block if you do not provide any arguments it is assumed the default and all platforms unless you define a collect\_data block specific for a platform.

# EXERCISE



## Reviewing the Ohai Gem

*We know where the plugins are located and what they look like. Now it's time to make one.*

**Objective:**

- ✓ Review the basic structure of the Ohai gem
- ✓ Review the 'language' plugin
- ✓ Review the 'python' plugin

We were able to view the contents of the gem and examine the contents of a few plugins to give us an understanding of how plugins are structured. Now it is time for use to create our own.

# DISCUSSION



## Discussion

How are the structures of a Rubygem and a Cookbook similar to each other?

What are the requirements when specifying the name of a Ohai plugin?

What is the difference between a Ruby Hash and a Mash?

# DISCUSSION



## Q&A

What questions can we answer for you?

What questions can we answer for you?



## Creating Ohai Plugins

# Objectives

After completing this module, you should be able to:

- Create a tested Ohai plugin

After completing this module you will be able to

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

### Objective:

- Define expectations for the Ohai plugin
- Create the Ohai plugin
- Define expectations for the recipe to deliver the Ohai plugin
- Create the recipe that delivers the Ohai plugin
- Define an integration test

## Installing the `chefspec-ohai` gem



```
> chef gem install chefspec-ohai
```

```
Successfully installed chefspec-ohai-0.1.1
1 gem installed
```

## Adding the Gem to the Spec Helper



~/apache/spec/spec\_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'
require 'chefspec/ohai'
```

## Creating a Directory for Plugins Tests



```
> mkdir ~apache/spec/unit/plugins
```

## Defining the First Expectation for Plugin

```
~/apache/spec/unit/plugins/apache_modules_spec.rb

require 'spec_helper'

describe_ohai_plugin :Apache do
  let(:plugin_file) { 'files/default/apache_modules.rb' }

  it 'provides apache/modules' do
    expect(plugin).to provides_attribute('apache/modules')
  end
end
```

## Executing the Tests to See Failure



```
> chef exec rspec
```

```
F.....
```

```
Failures:
```

```
1) Apache provides 'apache/modules'
```

```
Failure/Error: let(:plugin_source) { File.read(plugin_file) }
```

```
Errno::ENOENT:
```

```
No such file or directory @ rb_sysopen -  
files/default/apache_modules.rb
```

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

### Objective:

- ✓ Define expectations for the Ohai plugin
- Create the Ohai plugin
- Define expectations for the recipe to deliver the Ohai plugin
- Create the recipe that delivers the Ohai plugin
- Define an integration test

# Creating the Plugin as a Cookbook File



```
> chef generate file apache_modules.rb
```

```
Recipe: code_generator::cookbook_file
  * directory[/home/chef/apache/files/default] action create
    - create new directory /home/chef/apache/files/default
    - restore selinux security context
  * template[/home/chef/apache/files/default/apache_modules] action create
    - create new file /home/chef/apache/files/default/apache_modules
    - update content in file /home/chef/apache/files/default/apache_modules
from none to e3b0c4
```

## Defining the Plugin that Provides Values



~/apache/files/default/apache\_modules.rb

```
Ohai.plugin(:Apache) do
  provides 'apache/modules'
end
```

## Executing the Tests to See Success



```
> chef exec rspec
```

```
.....
```

```
Finished in 2.15 seconds (files took 1.88 seconds to load)
11 examples, 0 failures
```

## Defining an Expectation for Attribute Value

```
~/apache/spec/unit/plugins/apache_modules_spec.rb

require 'spec_helper'

describe_ohai_plugin :Apache do
  let(:plugin_file) { 'files/default/apache_modules.rb' }

  it 'provides apache/modules' do
    expect(plugin).to provides_attribute('apache/modules')
  end

  it 'correctly captures output' do
    allow(plugin).to receive(:shell_out).with('apachectl -t -D
DUMP_MODULES').and_return(double(stdout: 'OUTPUT'))
    expect(plugin_attribute('apache/modules')).to eq('OUTPUT')
  end
end
```

## Executing the Tests to See Failure



```
> chef exec rspec
```

```
.F.....
```

```
Failures:
```

```
1) Apache correctly captures output
Failure/Error: expect(plugin_attribute('apache/modules')).to
eq("OUTPUT")
```

```
NoMethodError:
```

```
undefined method `[]' for nil:NilClass
```

# Capturing the Apache Modules Content

```
~/apache/files/default/apache_modules.rb

Ohai.plugin :Apache do
  provides 'apache/modules'

  collect_data :default do
    apache(Mash.new)
    modules_cmd = shell_out('apachectl -t -D DUMP_MODULES')
    apache[:modules] = modules_cmd.stdout
  end
end
```

## Executing the Tests to See Success



```
> chef exec rspec
```

```
.....
```

```
Finished in 2.08 seconds (files took 1.88 seconds to load)
12 examples, 0 failures
```

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

### Objective:

- ✓ Define expectations for the Ohai plugin
- ✓ Create the Ohai plugin
- ❑ Define expectations for the recipe to deliver the Ohai plugin
- ❑ Create the recipe that delivers the Ohai plugin
- ❑ Define an integration test

## Adding a Dependency on the Ohai Cookbook

```
~/apache/metadata.rb

name 'apache'
maintainer 'The Authors'
maintainer_email 'you@example.com'
license 'All Rights Reserved'
description 'Installs/Configures apache'
long_description 'Installs/Configures apache'
version '0.1.0'
chef_version '>= 12.1' if respond_to?(:chef_version)

depends 'ohai'
# The `issues_url` points to the location where issues for this cookbook are
# tracked. A `View Issues` link will be displayed on this cookbook's page when
# uploaded to a Supermarket.
#
# issues_url 'https://github.com/<insert_org_here>/apache/issues'
```

## Creating the Recipe to Add the Plugin



```
> chef generate recipe ohai_apache_modules
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/apache/spec/unit/recipes] action create
    (up to date)
  * cookbook_file[/home/chef/apache/spec/spec_helper.rb] action
    create_if_missing (up to date)
  *
  template[/home/chef/apache/spec/unit/recipes/ohai_apache_modules_s
    pec.rb] action create_if_missing
  ...
...
```

## Delete the Context for Ubuntu

```
~/apache/spec/unit/recipes/ohai_apache_modules_spec.rb

require 'spec_helper'

describe 'apache::ohai_apache_modules' do
  context 'When all attributes are default, on Ubuntu 16.04' do
    let(:chef_run) do
      # for a complete list of available platforms and versions see:
      # https://github.com/customink/fauxhai/blob/master/PLATFORMS.md
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

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

# Defining the Expectation Plugin Deployed

```
~/apache/spec/unit/recipes/ohai_apache_modules_spec.rb
```

```
describe 'apache::ohai_apache_modules' do
  context 'When all attributes are default, on CentOS 6.9' do
    let(:chef_run) do
      # for a complete list of available platforms and versions see:
      # https://github.com/customink/fauxhai/blob/master/PLATFORMS.md
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.9')
      runner.converge(described_recipe)
    end

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

    it 'installs the modules plugin' do
      expect(chef_run).to create_ohai_plugin('apache_modules')
    end
  end
end
```

## Execute the Tests to See Failure



```
> chef exec rspec
```

```
.....F
```

```
Failures:
```

```
1) apache::ohai_apache_modules When all attributes are default,  
on an unspecified platform installs the modules plugin
```

```
Failure/Error: expect(chef_run).to  
create_ohai_plugin('apache_modules')
```

```
expected "ohai_plugin[apache_modules]" with action :create  
to be in Chef run. Other ohai_plugin resources:
```

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

### Objective:

- ✓ Define expectations for the Ohai plugin
- ✓ Create the Ohai plugin
- ✓ Define expectations for the recipe to deliver the Ohai plugin
- Create the recipe that delivers the Ohai plugin
- Define an integration test

## Defining the Ohai Plugin in the Recipe

```
~/apache/recipes/ohai_apache_modules.rb

#
# Cookbook:: apache
# Recipe:: ohai_apache_modules
#
# Copyright:: 2018, The Authors, All Rights Reserved.
ohai_plugin 'apache_modules'
```

We are using a Custom Resource from the Ohai cookbook to provide the apache\_modules attribute.

## Executing the Tests to See Success



```
> chef exec rspec
```

```
.....
```

```
Finished in 3.33 seconds (files took 1.87 seconds to load)
14 examples, 0 failures
```

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

**Objective:**

- ✓ Define expectations for the Ohai plugin
- ✓ Create the Ohai plugin
- ✓ Define expectations for the recipe to deliver the Ohai plugin
- ✓ Create the recipe that delivers the Ohai plugin
- Define an integration test

# Appending the New Recipe to the Suite

```
~/apache/.kitchen.yml

# ... TOP OF KITCHEN CONFIGURATION ...
suites:
  - name: default
    run_list:
      - recipe[apache::default]
      - recipe[apache::ohai_apache_modules]
    verifier:
      inspec_tests:
        - test/smoke/default
    attributes:
```

# Converging the Updated Default Suite



```
> kitchen converge
```

```
-----> Starting Kitchen (v1.19.2)
-----> Converging <default-centos-69>...
    Preparing files for transfer
    Preparing dna.json
    Resolving cookbook dependencies with Berkshelf 6.3.1...
    Removing non-cookbook files before transfer
    Preparing validation.pem
    Preparing client.rb
-----> Chef Omnibus installation detected (install only if missing)
    Transferring files to <default-centos-69>
    Starting Chef Client, version 13.8.5
      resolving cookbooks for run list: ["apache::default", "apache::ohai_apache_modu
les"]
      Synchronizing Cookbooks:
```

## Defining an Expectation for the Plugin

```
~/apache/test/smoke/default/ohai_apache_modules.rb

plugin_directory = '/tmp/kitchen/ohai/plugins'

describe command("ohai -d #{plugin_directory} apache") do
  its(:stdout) { should match(/core_module/) }
end
```

## Verifying the New Expectation



```
> kitchen verify
```

```
Target: ssh://kitchen@localhost:32769
```

```
Command curl
```

```
✓ http://localhost stdout should match /Welcome users/
```

```
Command curl
```

```
✓ http://localhost:8080 stdout should match /Welcome admins/
```

```
Command ohai
```

```
✓ -d /tmp/kitchen/ohai/plugins apache stdout should match /core_module/
```

```
Test Summary: 3 successful, 0 failures, 0 skipped
```

# EXERCISE



## Creating an Ohai Plugin

*Being able to learn more about our nodes will make our reporting more powerful and benefit the recipes we write.*

### Objective:

- ✓ Define expectations for the Ohai plugin
- ✓ Create the Ohai plugin
- ✓ Define expectations for the recipe to deliver the Ohai plugin
- ✓ Create the recipe that delivers the Ohai plugin
- ✓ Define an integration test

# DISCUSSION



## Q&A

What questions can we answer for you?



# Tuning Ohai

# Objectives

After completing this module, you should be able to:

- Describe how you configure the node to automatically load ohai plugins
- Describe how to enable ohai hints
- Describe how to remove plugins that you do not want executed
- Describe where you can find more information about better Ohai performance

After completing this module you will be able to

# EXERCISE



## Run Ohai Smoother

*There are a few more things to learn about Ohai that could help increase performance.*

**Objective:**

- Configure the node to automatically load ohai plugins
- Enable ohai hints
- Remove plugins that you do not want executed
- Choose only the plugins you want executed

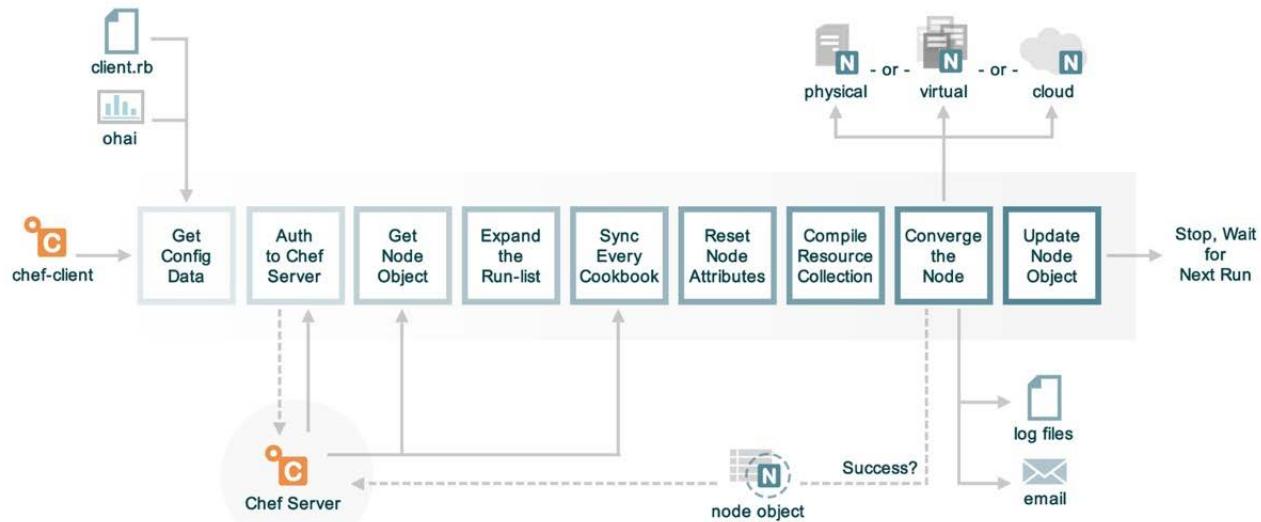
# CONCEPT



## Node Configuration File

All nodes have a configuration file that contain details about node, overrides, and other data. This is the client.rb file on the node.

# The Anatomy of a chef-client Run



# CONCEPT



## chef-client cookbook

The chef-client cookbook contains a number of useful recipes:

- **service (default)**  
Run chef-client as a service, converging at a periodic interval
- **delete\_validation**  
Remove the organization validation key [SECURITY ISSUE]
- **config**  
Define node configuration

<https://supermarket.chef.io/cookbooks/chef-client>

# Viewing the chef-client config Recipe

```
~/cookbooks/chef-client/recipes/config.rb
```

```
# ... OTHER RESOURCES ...
template "#{node['chef_client']['conf_dir']}/client.rb" do
  source 'client.rb.erb'
  owner d_owner
  group node['root_group']
  mode '0644'
  variables(
    chef_config: node['chef_client']['config'],
    chef_requires: chef_requires,
    ohai_disabled_plugins: node['ohai']['disabled_plugins'],
    start_handlers: node['chef_client']['config']['start_handlers'],
    report_handlers: node['chef_client']['config']['report_handlers'],
    exception_handlers: node['chef_client']['config']['exception_handlers']
  )
}
```

Source code for chef-client cookbook: <https://github.com/chef-cookbooks/chef-client>

Link to chef-client::config.rb <https://github.com/chef-cookbooks/chef-client/blob/master/recipes/config.rb>

```
~/cookbooks/chef-client/templates/default/client.rb.erb

# ... OTHER STATEMENTS ...

<% unless node["ohai"]["plugin_path"].nil? -%>
ohai.plugin_path << "<%= node['ohai']['plugin_path'] %>"
<% end -%>
<% unless @ohai_disabled_plugins.empty? -%>
ohai.disabled_plugins = [<%= @ohai_disabled_plugins.map { |k| k.match(/:/) ? k.gsub(/^:/, '')}.to_sym.inspect : k.inspect ].join(",") %>
<% end -%>

# ... OTHER STATEMENTS ...
```

# PROBLEM



## The Work

- Add the chef-client cookbook to your cookbook collection
- Provide attributes in a wrapper cookbook, role, or environment for:

```
node['ohai']['plugin_path']
```

- Add chef-client cookbook to every node's run list

# EXERCISE



## Run Ohai Smoother

*There are a few more things to learn about Ohai that could help increase performance.*

**Objective:**

- ✓ Configure the node to automatically load ohai plugins
- Enable ohai hints
- Remove plugins that you do not want executed
- Choose only the plugins you want executed

# CONCEPT



## Not all Node Plugins are Executed

Some of the plugins will not automatically run against your node. This is particularly true of the cloud provider plugins. As the node does not often know the environment in which it is being run.

# CONCEPT



## Ohai Hints

For those plugins that are not executed you can leave them a hint file that will ensure they are executed.

# PROBLEM



## The Work

- Add the ohai cookbook to your cookbook collection
- Define a recipe that uses the ohai cookbook's ohai\_hint resource
- Add this recipe to every node's run list

<https://github.com/chef-cookbooks/ohai>

# PROBLEM



## Custom Resources

### `ohai_hint`

Creates Ohai hint files, which are consumed by Ohai plugins in order to determine if they should run or not.

#### Resource Attributes

- `hint_name` - The name of hints file and key. Should be string, default is name of resource.
- `content` - Values of hints. It will be used as automatic attributes. Should be Hash, default is empty Hash
- `compile_time` - Should the resource run at compile time. This defaults to true

#### Examples

Hint file installed to the default directory:

```
ohai_hint 'ec2'
```

# EXERCISE



## Run Ohai Smoother

*There are a few more things to learn about Ohai that could help increase performance.*

**Objective:**

- ✓ Configure the node to automatically load ohai plugins
- ✓ Enable ohai hints
- ❑ Remove plugins that you do not want executed
- ❑ Choose only the plugins you want executed



~/cookbooks/chef-client/templates/default/client.rb.erb

```
<% unless node["ohai"]["plugin_path"].nil? -%>
ohai.plugin_path << "<%= node['ohai']['plugin_path'] %>"
<% end -%>
<% unless @ohai_disabled_plugins.empty? -%>
ohai.disabled_plugins = [<%= @ohai_disabled_plugins.map { |k| k.match(/:/) ? k.gsub(/^:/, '') }.to_sym.inspect : k.inspect ].join(",")
<% end -%>
```

<https://github.com/chef-cookbooks/chef-client/blob/master/templates/default/client.rb.erb>

# Viewing the chef-client config Recipe

```
~/cookbooks/chef-client/recipes/config.rb

template "#{node['chef_client']['conf_dir']}/client.rb" do
  source 'client.rb.erb'
  owner d_owner
  group node['root_group']
  mode '0644'
  variables(
    chef_config: node['chef_client']['config'],
    chef_requires: chef_requires,
    ohai_disabled_plugins: node['ohai']['disabled_plugins'],
    start_handlers: node['chef_client']['config']['start_handlers'],
    report_handlers: node['chef_client']['config']['report_handlers'],
    exception_handlers: node['chef_client']['config']['exception_handlers']
  )

```

# PROBLEM



## The Work

- Add the chef-client cookbook to your cookbook collection
- Select attributes you want to remove
- Find the name of the plugin that provides those attributes
- Provide attributes in a wrapper cookbook, role, or environment for:

```
node['ohai']['disabled_plugins']
```

- Add chef-client cookbook to every node's run list

# EXERCISE



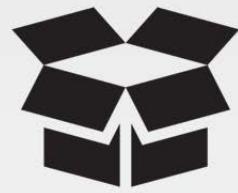
## Run Ohai Smoother

*There are a few more things to learn about Ohai that could help increase performance.*

**Objective:**

- ✓ Configure the node to automatically load ohai plugins
- ✓ Enable ohai hints
- ✓ Remove plugins that you do not want executed
- ❑ Choose only the plugins you want executed

# CONCEPT



## Whitelist Node Attributes

When it seems like you are disabling far too many plugins and you want to consider attacking it from the other side:

<https://docs.chef.io/ohai.html#whitelist-attributes>

# EXERCISE



## Run Ohai Smoother

*There are a few more things to learn about Ohai that could help increase performance.*

**Objective:**

- ✓ Configure the node to automatically load ohai plugins
- ✓ Enable ohai hints
- ✓ Remove plugins that you do not want executed
- ✓ Choose only the plugins you want executed

# DISCUSSION



## Q&A

What questions can we answer for you?

