CHEF™

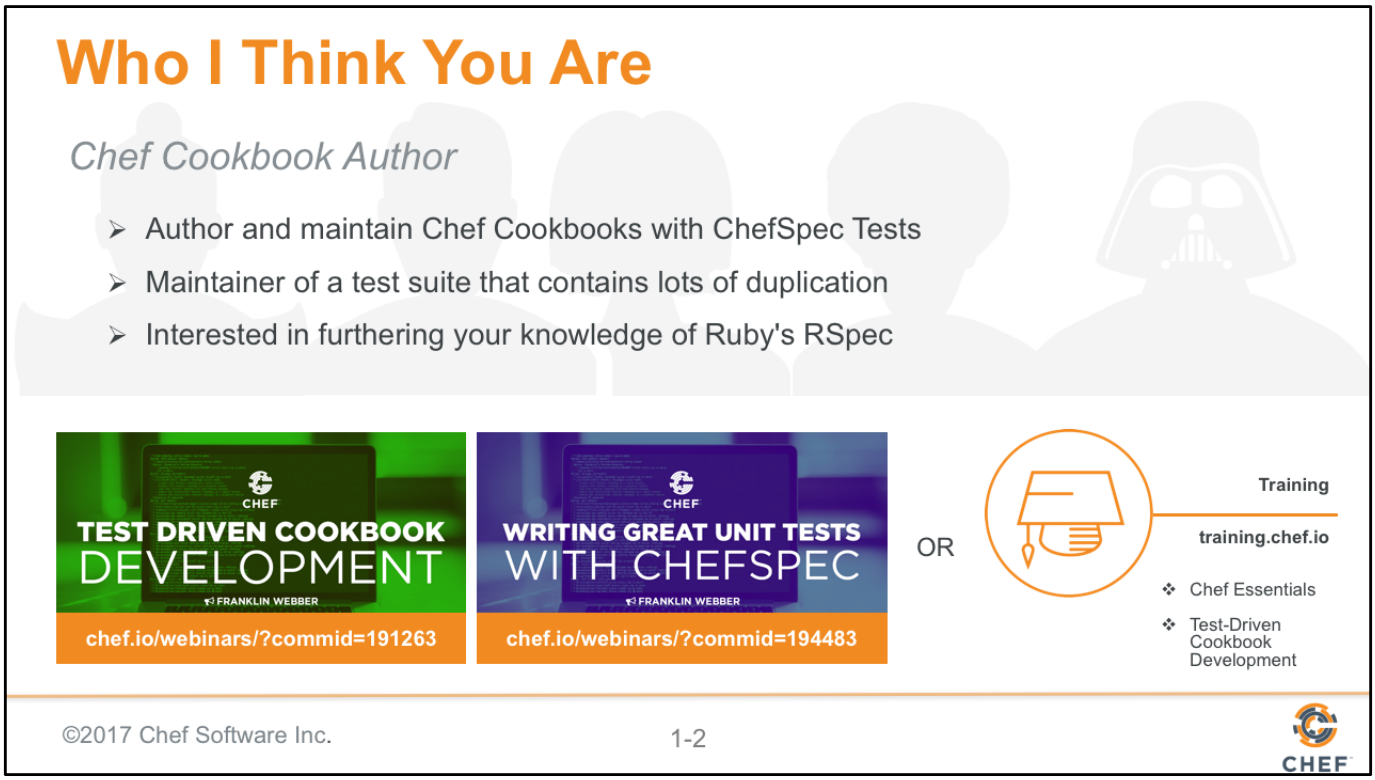
Chef Training Services

Elegant Tests

Guide

Course Rev 1.0.0

Welcome to Elegant Tests.



When creating this workshop we spent a lot of time thinking about who you are to help guide us in preparing the best content to assist you and to define the scope of what to share with you.

First and foremost we made the assumption that you are an author or co-author of cookbooks. By that we mean you have edited or created recipes and defined attributes. During your time working with Chef you have written unit tests to help describe the intended purpose of your recipes. Ideally we are talking to some of you that have created a bit of a problem for yourself with how big your test suite has become. And whether or not you have that problem you are interested in learning more about how Ruby's RSpec can increase the elegance of the tests you write.

Introduce Yourselves

- Name
- Current Role
- Previous Roles / Background
- Experience with Chef

Conversation Topics (complete these sentences)

The most difficult thing that I've tested was ...

My team and I review code before we commit it to production by ....

Before we begin let's take a moment to get to know one another.

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

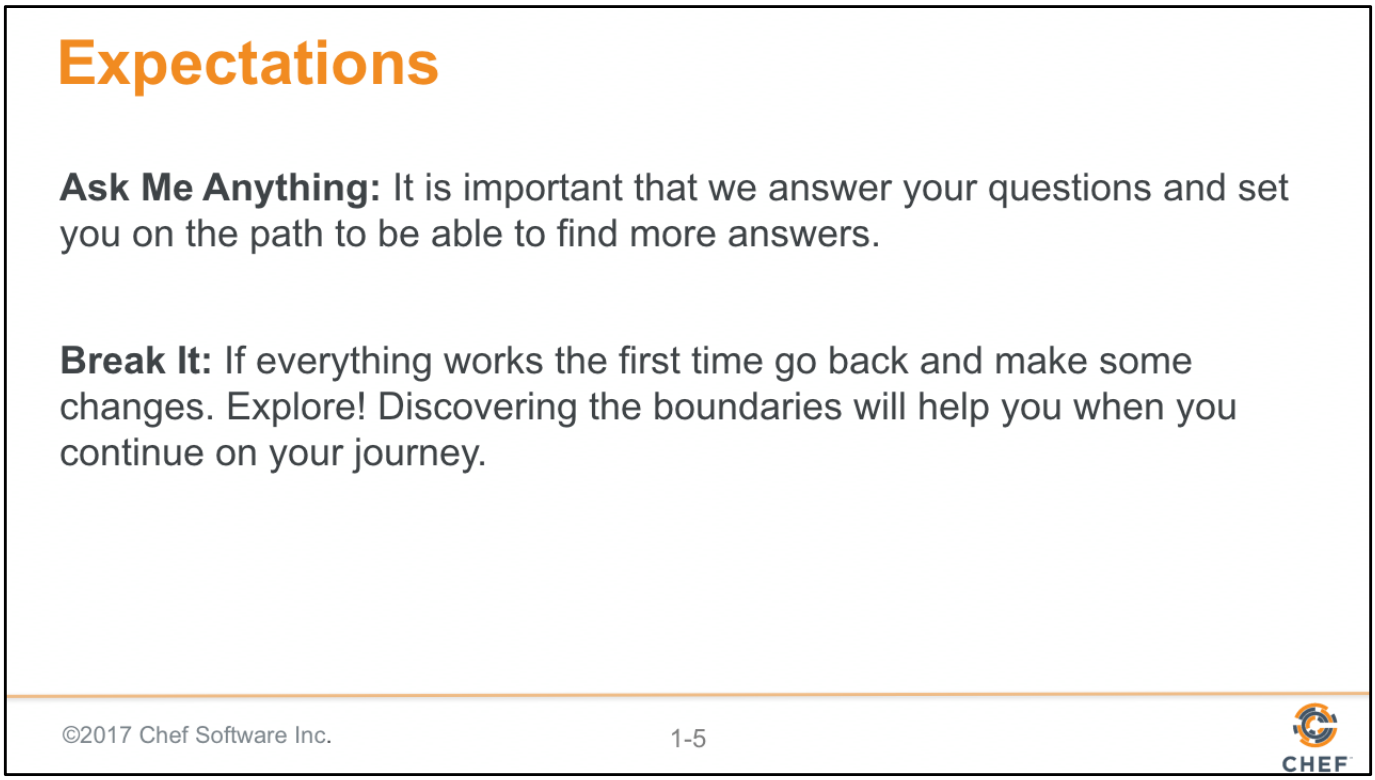
You will leave this class with the ability to combat redundancy in your tests.

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 combat the redundancy in your test suite and then extract those helpers into a Ruby gem.

Chef is built on top of Ruby. The testing tools built on top of RSpec. 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.



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.

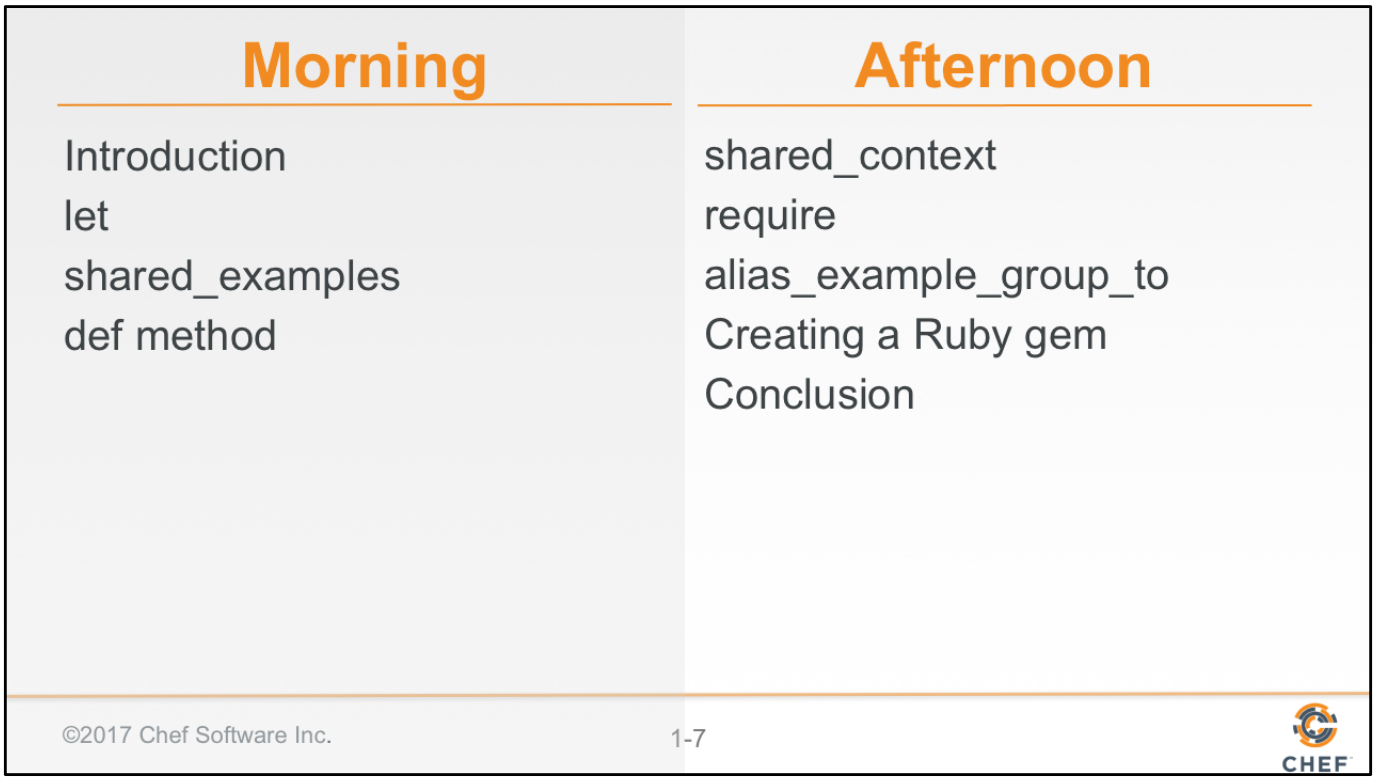
This curiosity can also be employed by exploring the boundaries of the tools you are using and the language you are writing. The exercises and the labs we will perform will often lead you through examples that work from the beginning to the end. When you develop solutions it is rare that something works 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

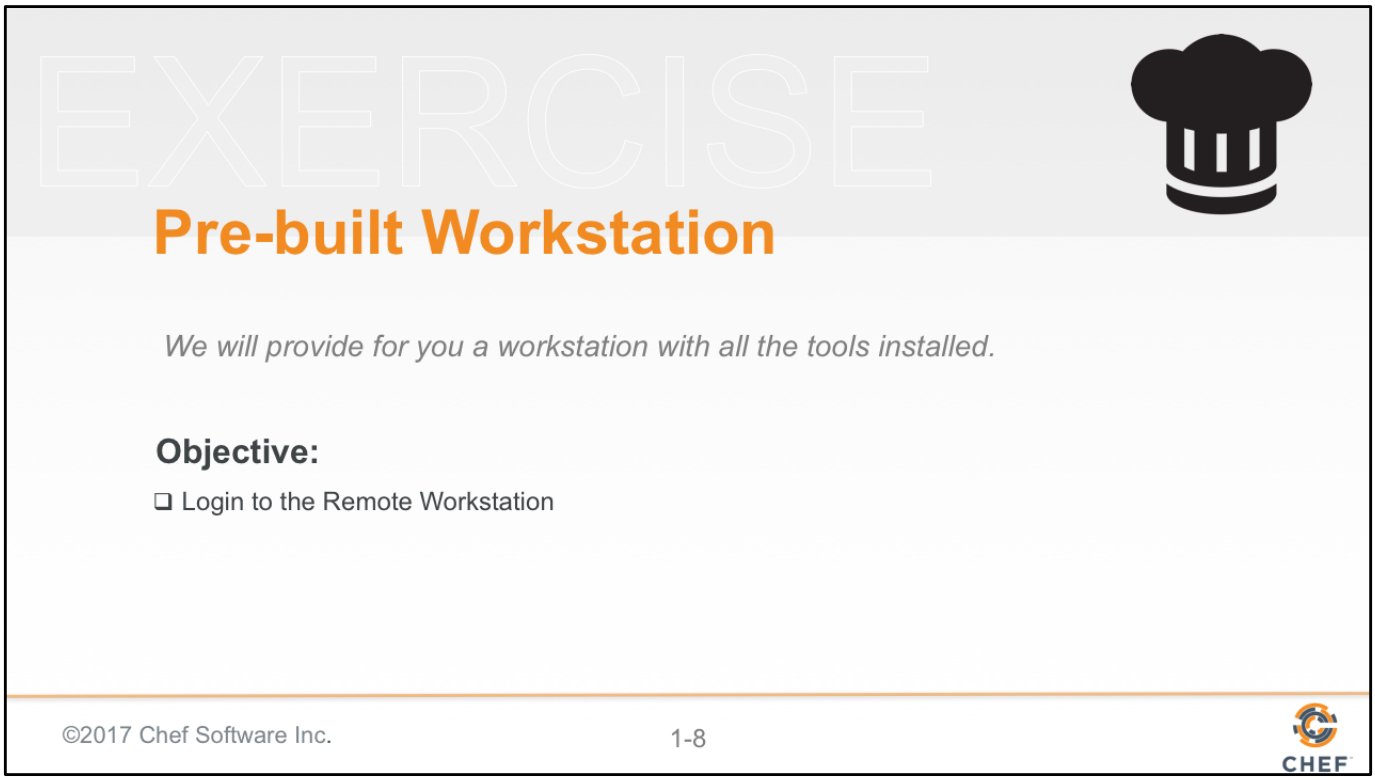
This course is designed to be hands on. You will run lots of commands, write lots of code, and express your understanding.

- **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.

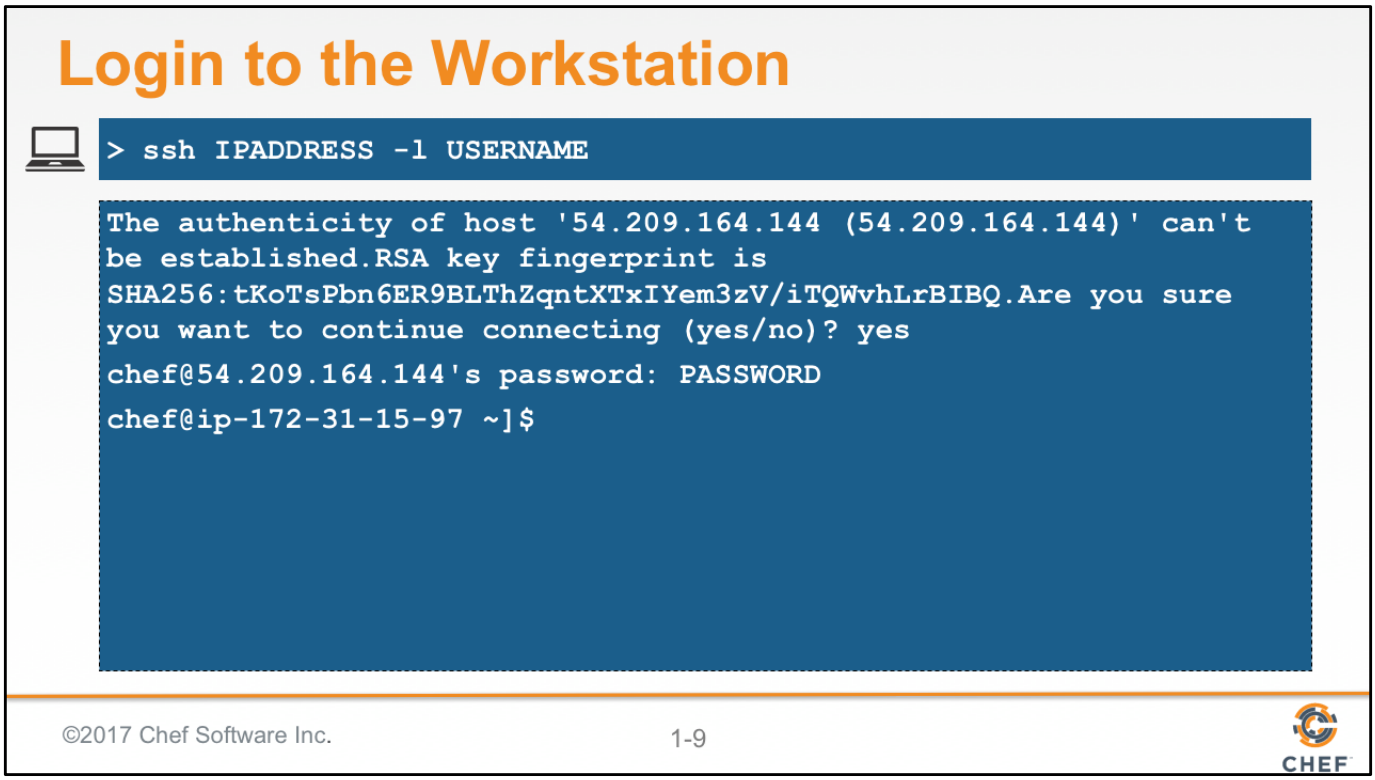


This is the outline of the events for this workshop. Please take a moment to review this list to ensure that the topics listed here meet your expectations. Take a moment to note which topics are of most interest to you. Also note which topics are not present here on this list. We will discuss your thoughts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



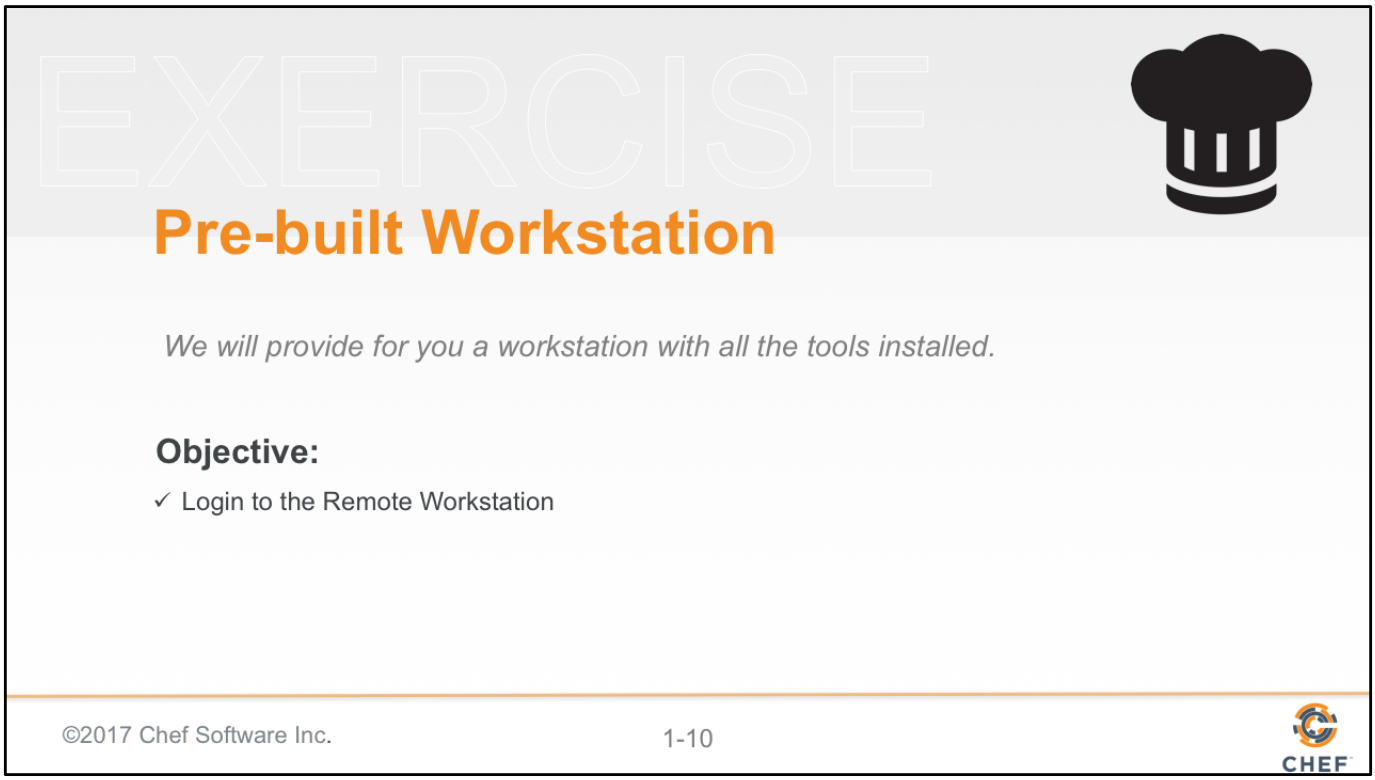
As I mentioned there is a lot work planned for the day. To ensure we focus on the concepts we introduce and not on troubleshooting systems we are providing you a workstation with the necessary tools installed to get started right away.

Instructor Note: The learners for this workshop can use their own workstations. However, if there are any troubles or concerns it is important to provide them with one to ensure that the time is spent on learning the material and not troubleshooting any installation issues. 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.

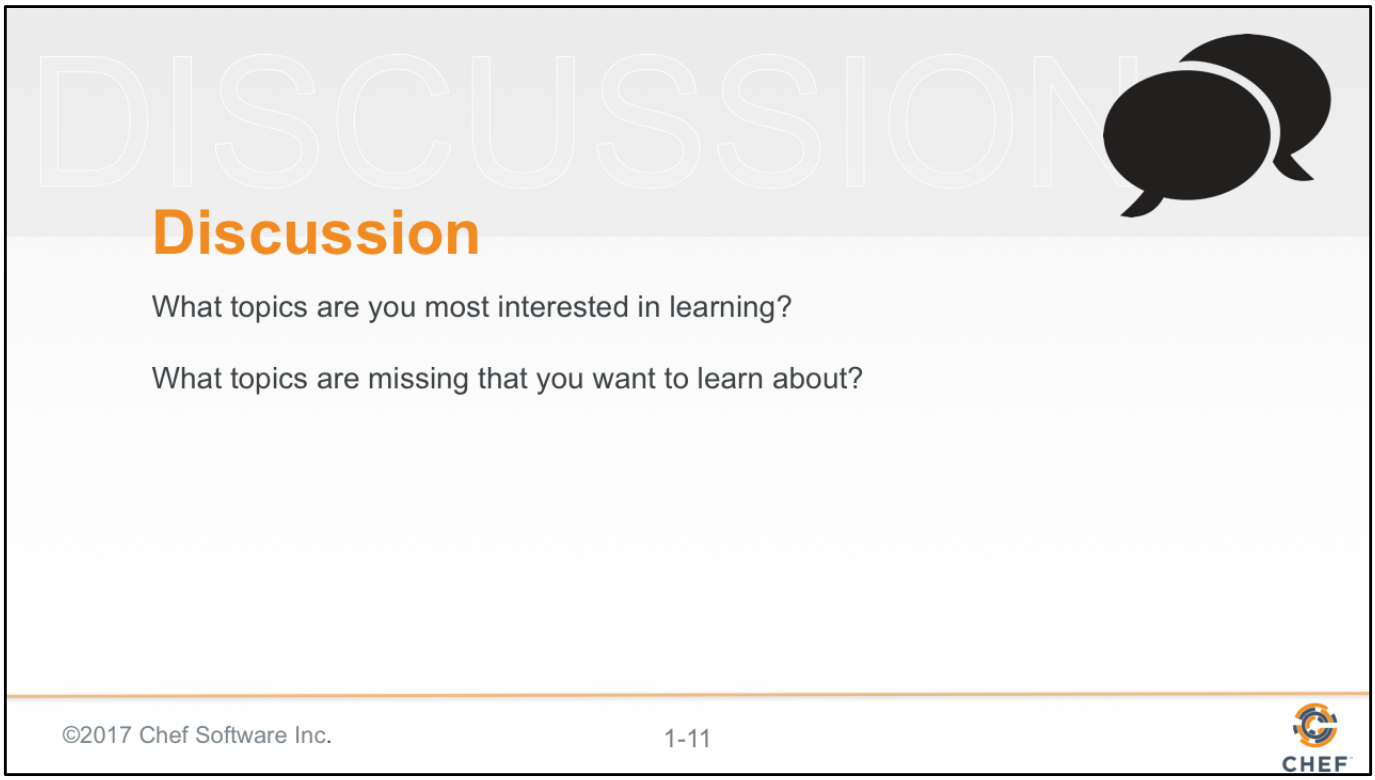


I will provide you with the address, username and password of the workstation. With that information you will need to use the SSH tool that you have installed to connect that workstation.

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.

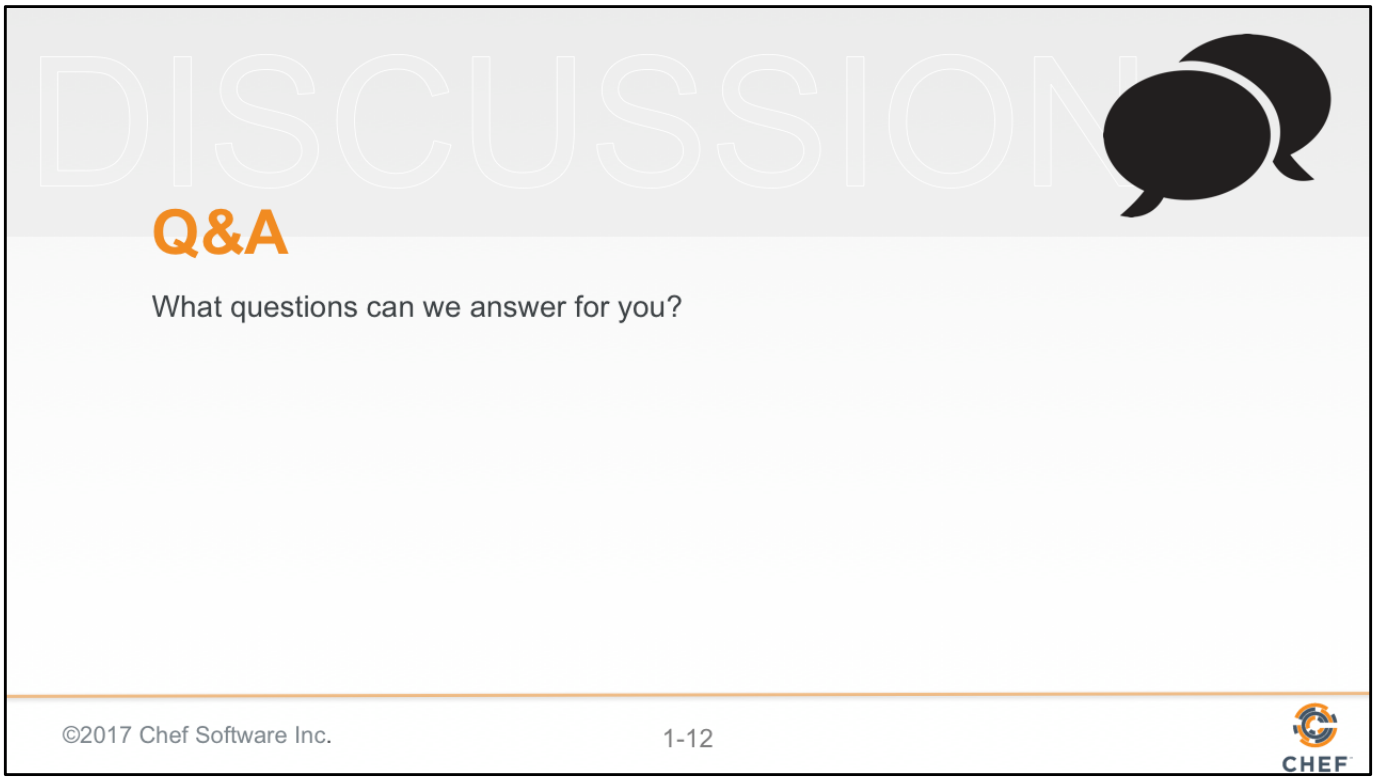


Now that you are connected to that workstation we have taken care of all the necessary work to get started with the training.



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.



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


CHEF™

It is important that before we make changes we understand the work that we are attempting to accomplish; the problem we want to solve. To understand the work we are going to accomplish here we need to spend some time reviewing the existing body of work.

This code review process will help us understand the problem that the code attempts to solve, how it solves the problem, and allow us to start thinking about ways in which we could make that solution easier to understand and maintain.

Objective

The objective of this section is to read the code that I have provided to you. Evaluate the code to understand its purpose and then judge how effective the code was in reaching that goal.

code review

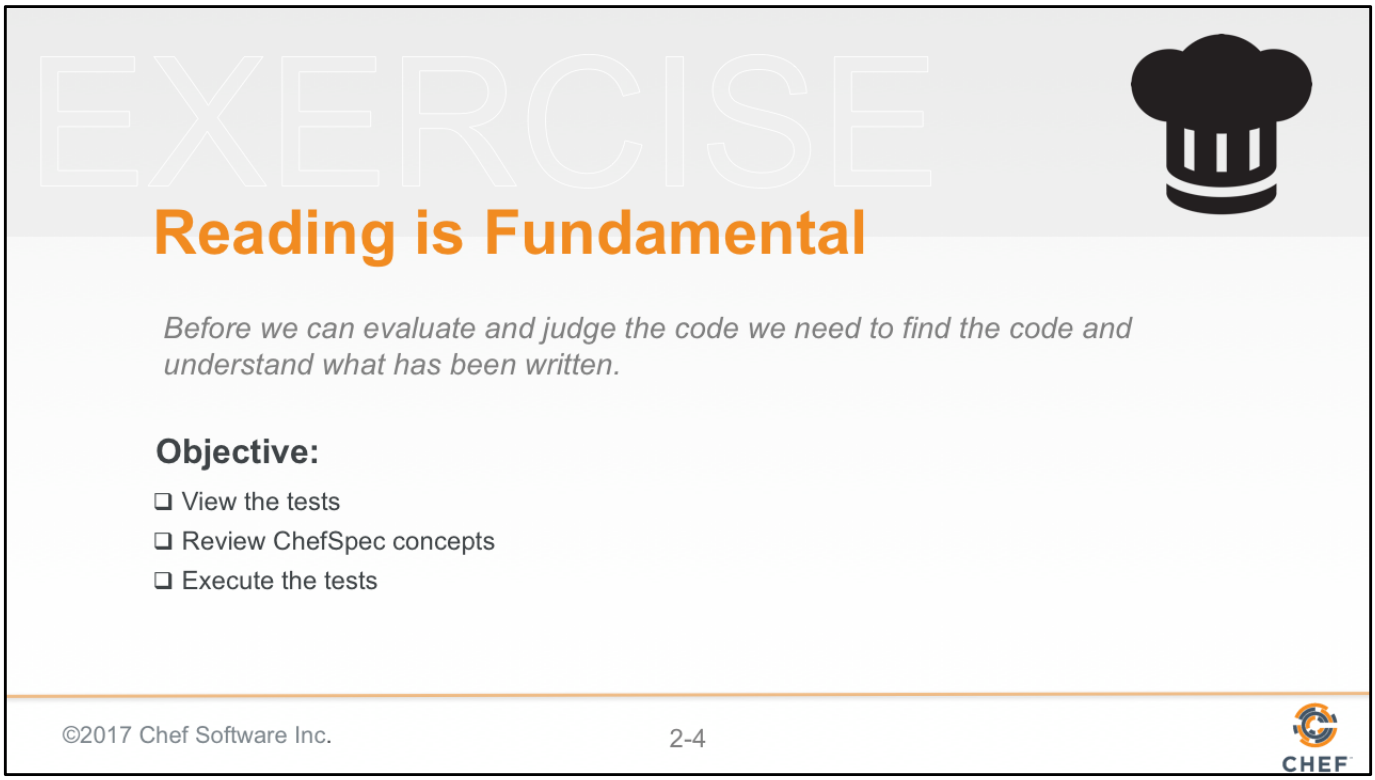
I finished all the work on the cookbook!

Heading out on paternity leave. Good Luck!

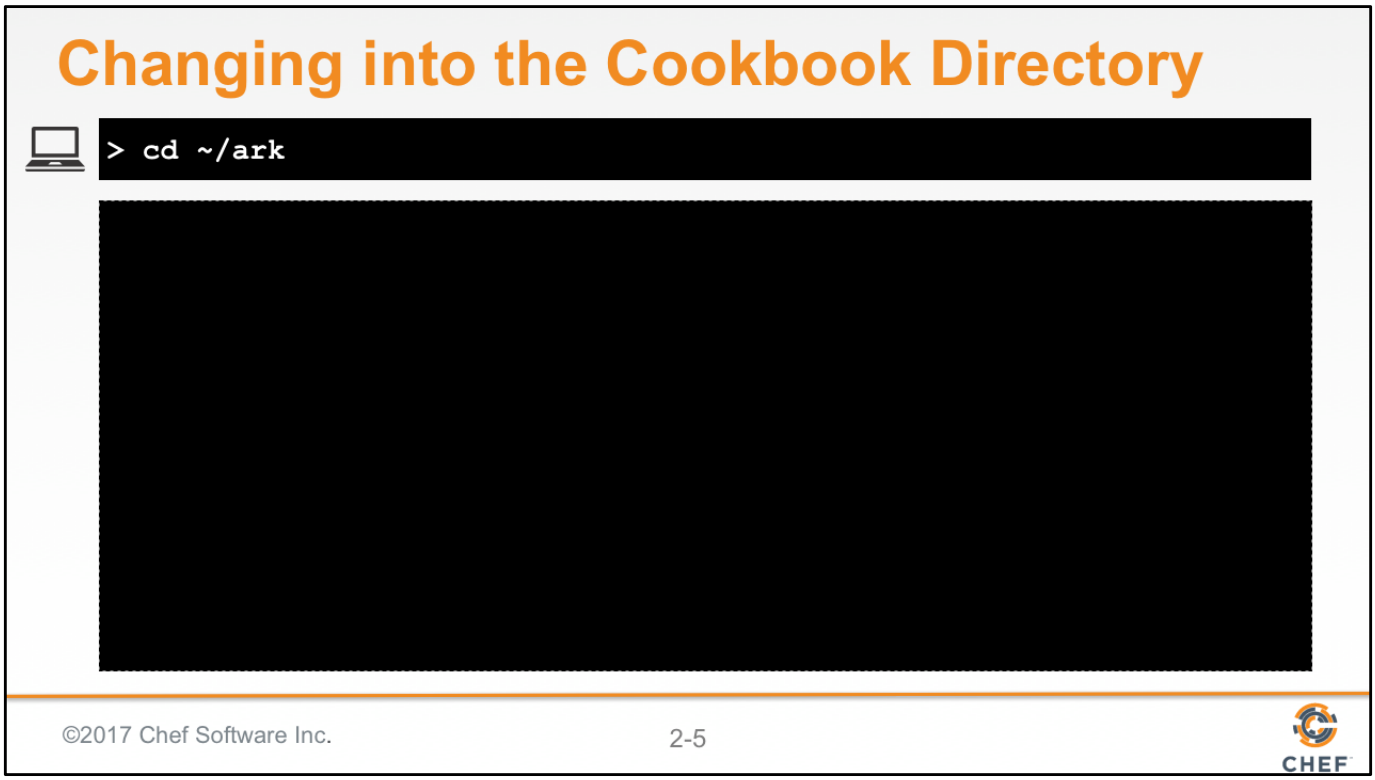
Sure! I'll take a look at it!

CHEF

During the introduction you may have shared some of the many ways in which you review code. Code reviews take many different forms within a team and within organizations.



To do this code review right we need to examine the code within the cookbook, review some ChefSpec concepts and then execute the tests to see if the cookbook is in working order.



Change into the cookbook directory.

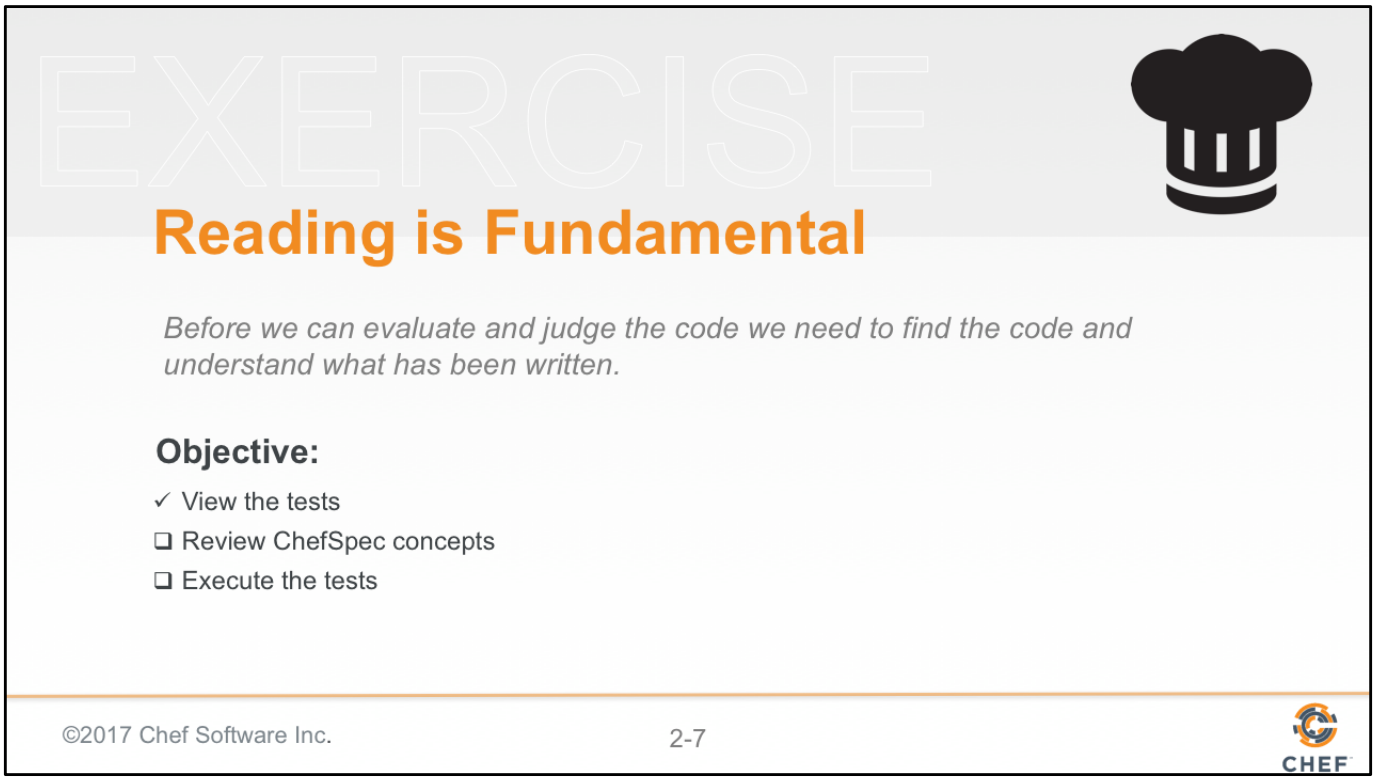
Viewing the Recipe Specification

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

```
require 'spec_helper'

describe 'ark::default' do
  context 'when no attributes are specified, on an unspe...orm' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
    end

    it 'installs necessary packages' do
```


CHEF

Open up the following file within your editor. This is the test file that defines the expectations for the default recipe.



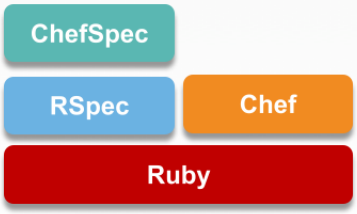
Now it is time to review some ChefSpec concepts.

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.

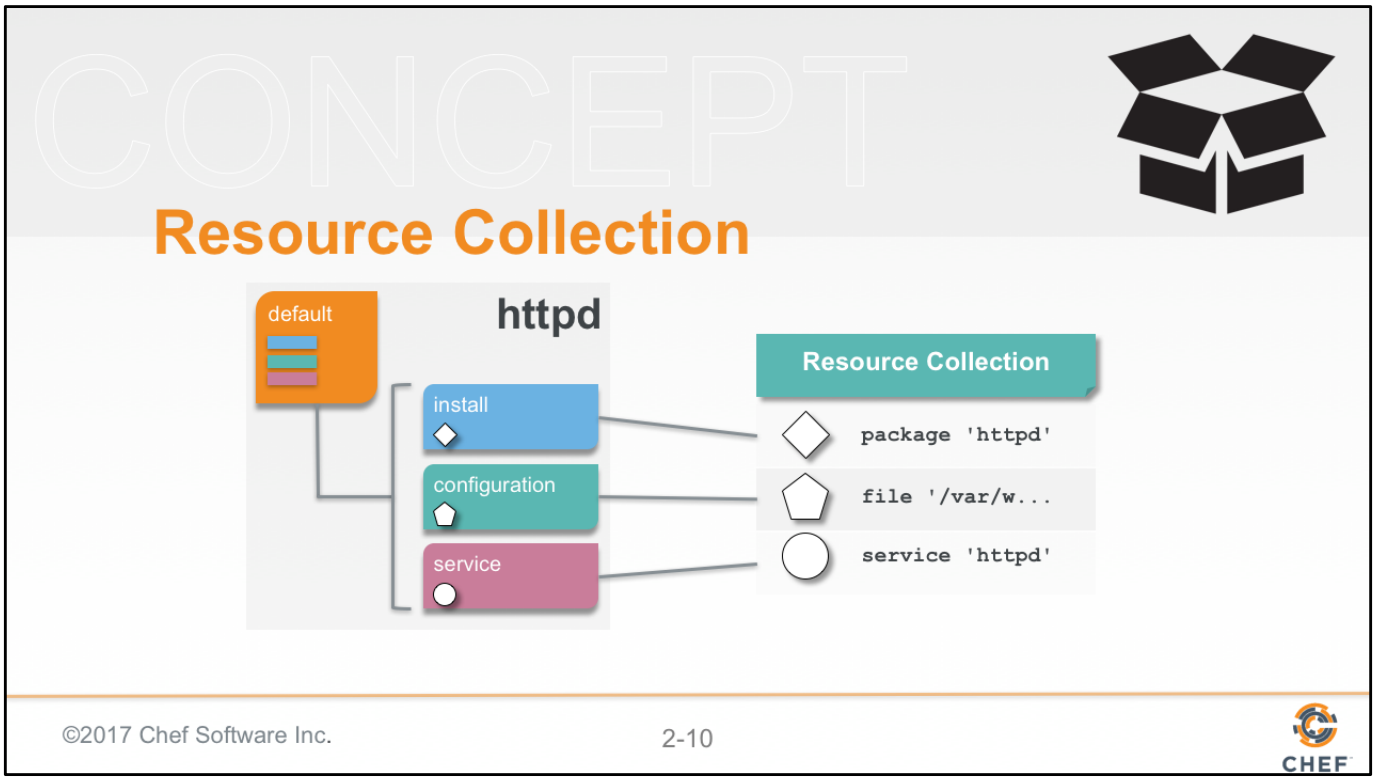
ChefSpec provides helpers and tools that allow you to express expectations about the state of **resource collection**.



CHEF

ChefSpec provides a method for us to create an in-memory execution that builds the resource collection, and then setting up expectations about the state of the resource collection. ChefSpec, similar to InSpec or ServerSpec, 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.

Testing with ChefSpec is different than testing with Test Kitchen because you are focused on testing the state of the resource collection. With ChefSpec tests you ignore the underlying hardware, installing Chef, and applying the run list to a real system.



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.

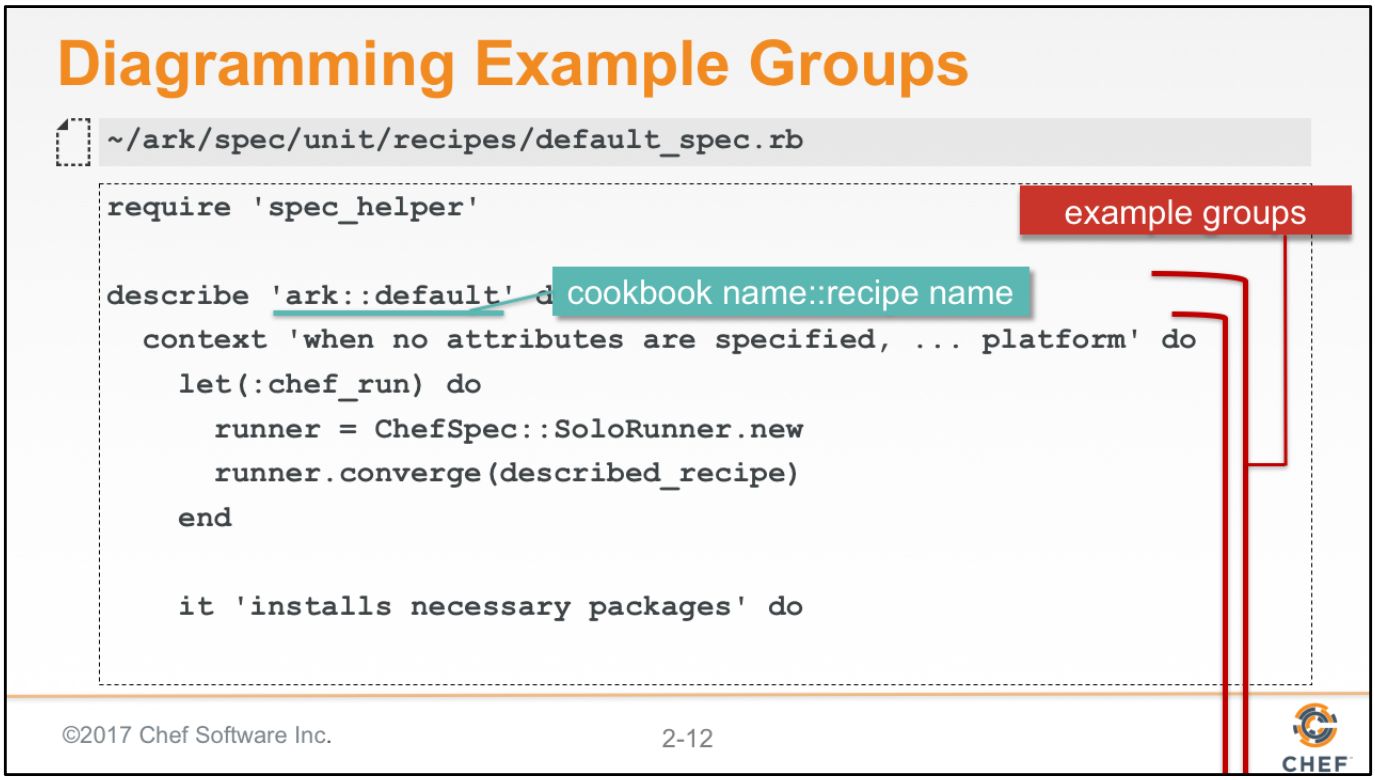
The 'Resource Collection' is almost like a to-do list for the node. It contains the list of all the resources, in order, that need to be accomplished to bring the instance into the desired state. Later, in the converge step, the resources defined in the Resource Collection are executed and perform their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

Viewing the Recipe Specification

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

```
require 'spec_helper'

describe 'ark::default' do
  context 'when no attributes are specified, on an unspe...orm' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
    end

    it 'installs necessary packages' do
```

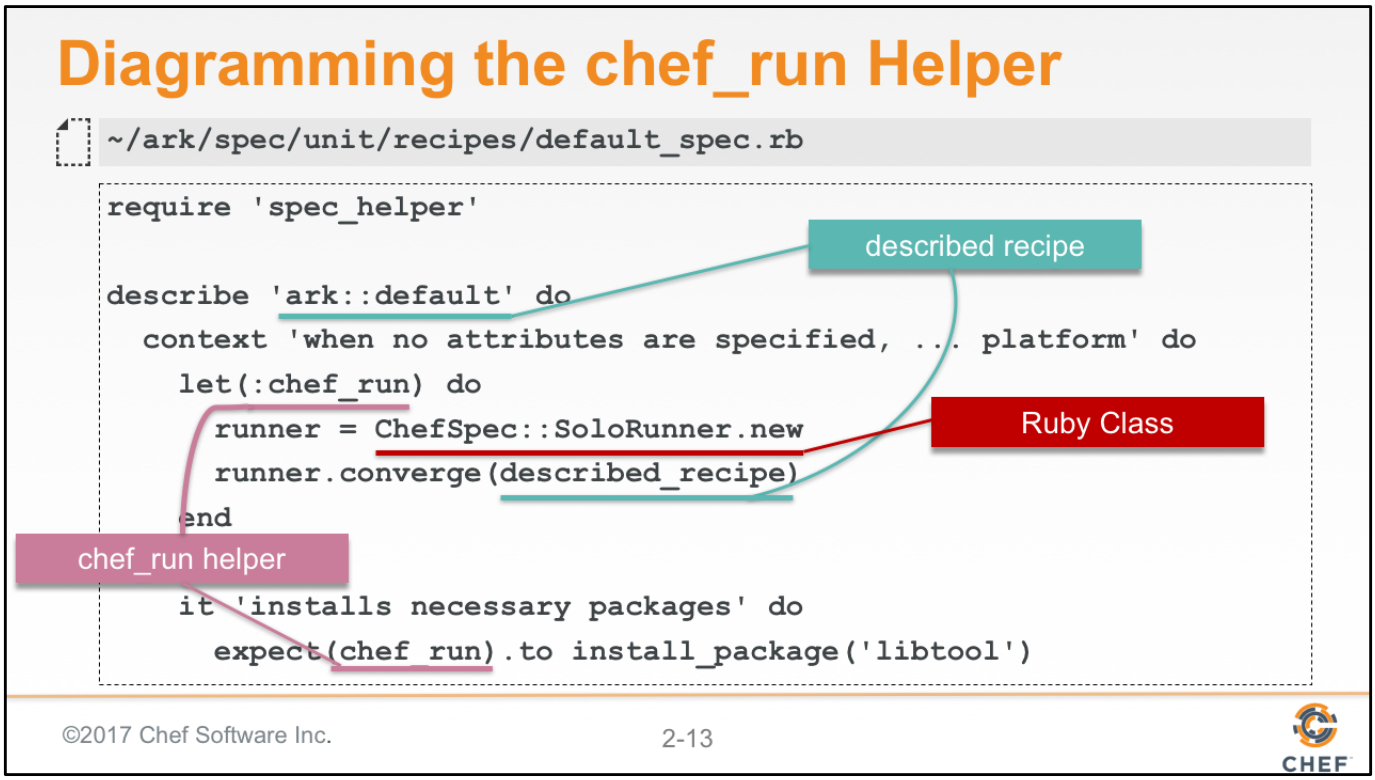

Here again is the default specification file. Over the next few slides let's break down the various concepts employed.



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 httpd 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 an unspecified platform.

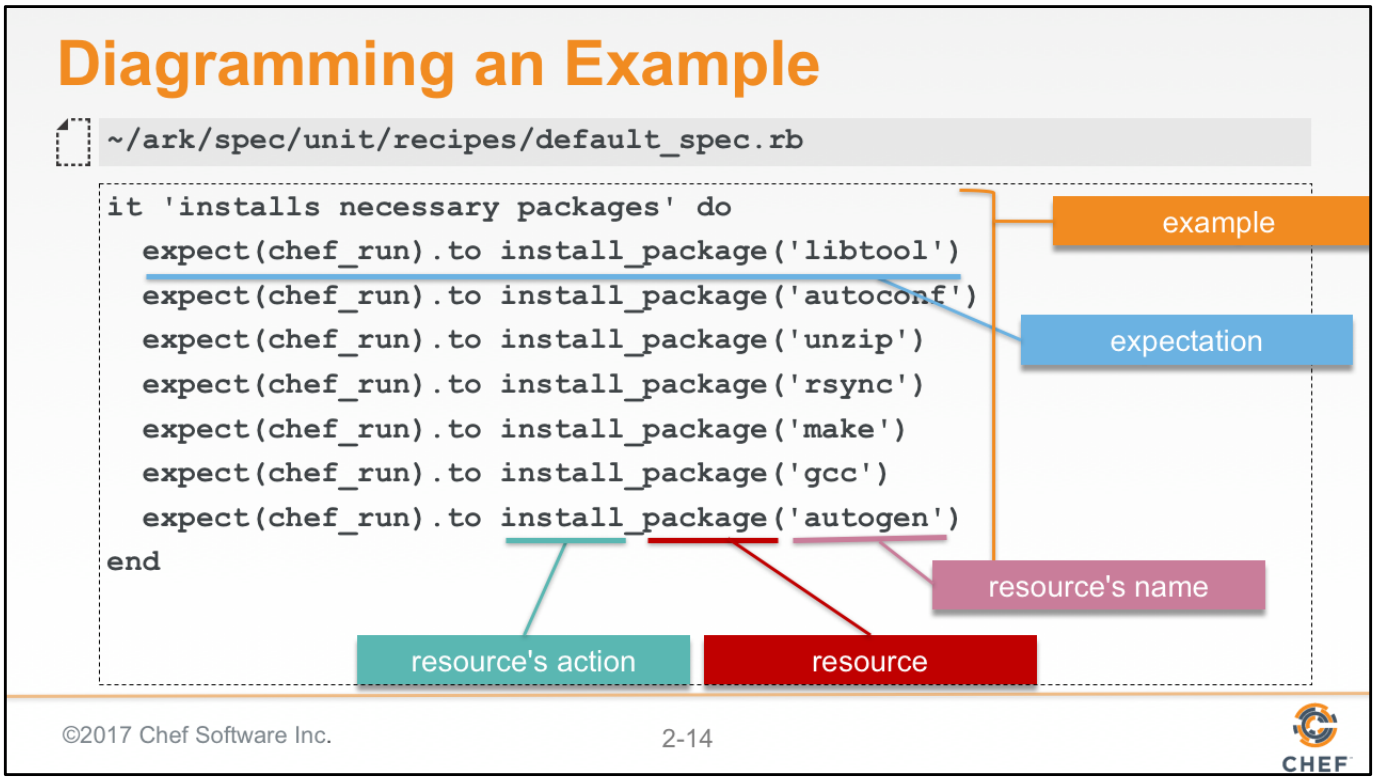
Note: 'describe' and 'context' are almost completely interchangeable with one exception. 'context' cannot be used as the outermost example group. 'context' is often used to help articulate a particular scenario like simulating a platform, the different path the code may take because of a node attribute or the result of some system state or calculation.



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 'SoloRunner' 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.



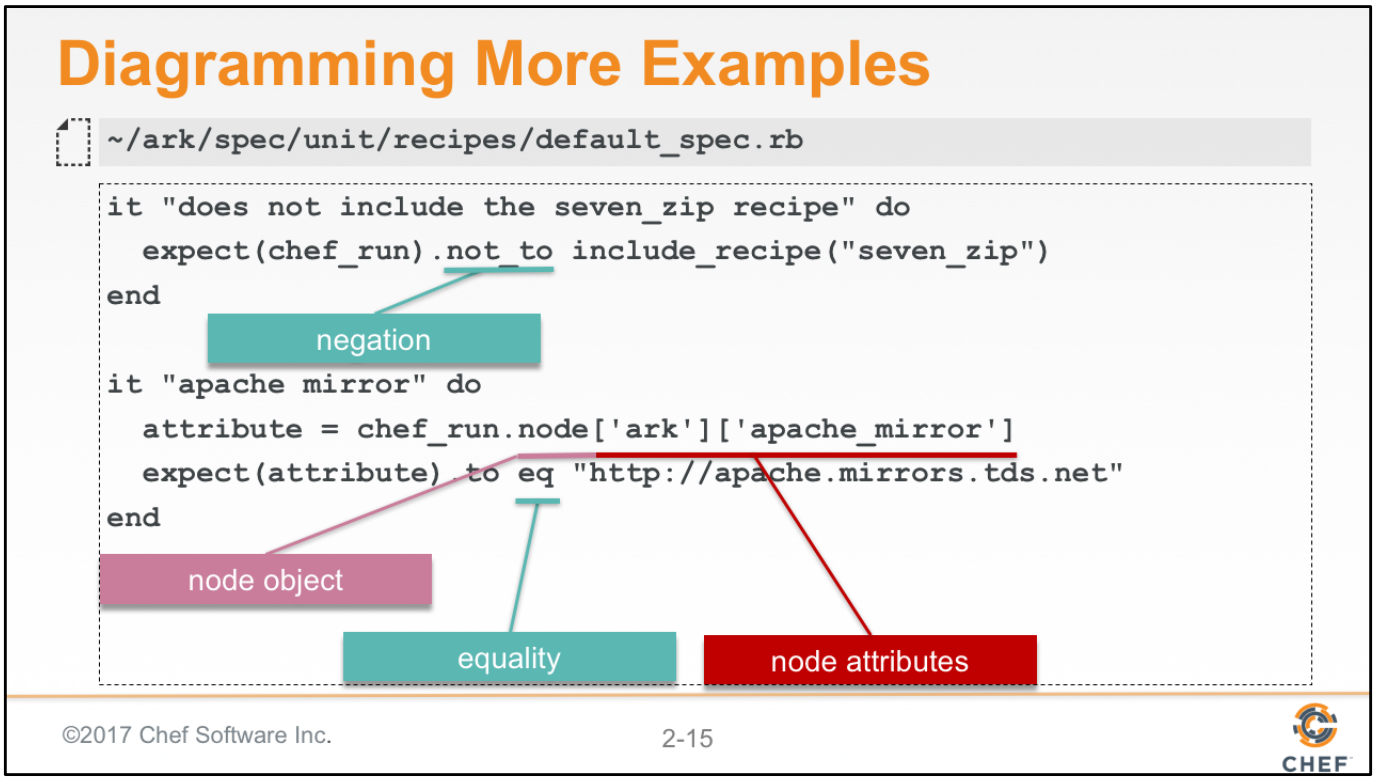
Within the inner context we finally set the stage for us to define our examples with their expectations.

The first example asserts that the 'necessary packages' have been installed in the system. Within that example we see multiple expectations.

Each expectation is again attempting to use a more natural language way to express the state of the system. Read aloud you might see this as "I expect the chef run to install the package libtool."

Each expectation generally starts the same. We want to ensure that the value that is wrapped within the expect method to meet the requirements setup by the matcher defined on the right.

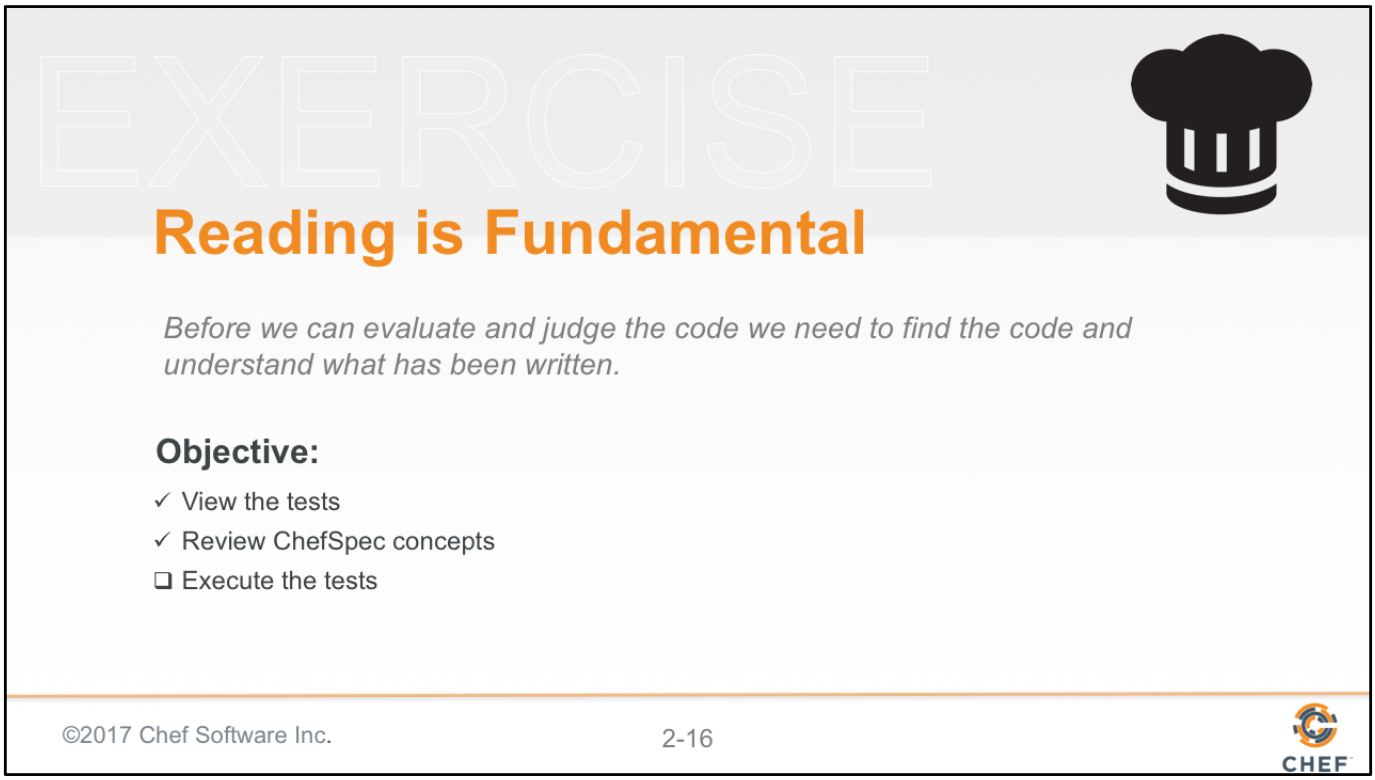
The matcher in this case, install_package('libtool'), is created from the resource type, the action the resource takes and the name of that resource.



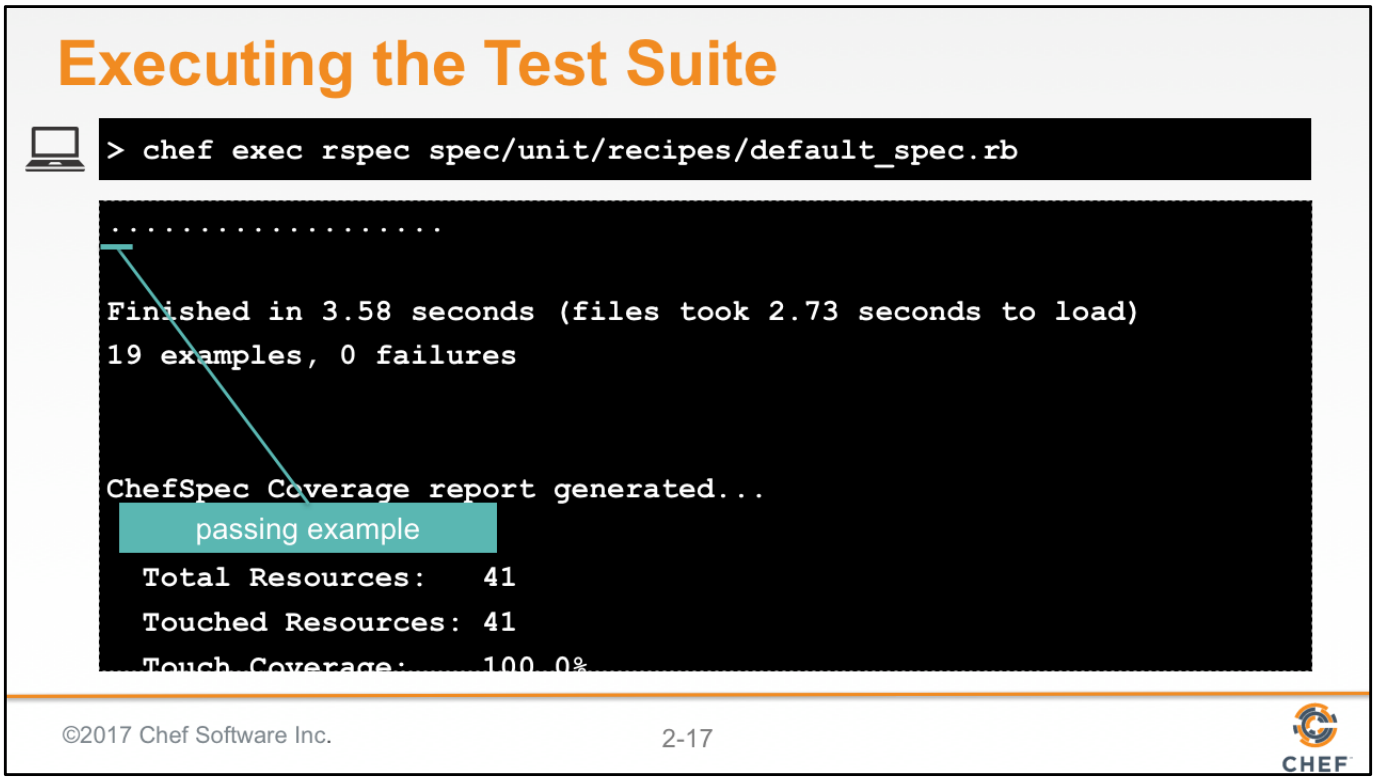
Expectations can also state what something is not and that is done with a negation. Instead of using the 'to', we use 'not_to'. This is useful in this first example show here as we are ensuring that by default we are not including a recipe that should only be included when the context states this recipe is being converged on a Windows system.

Along with checking to see if resources take the appropriate action or the appropriate recipe is included we can also access the node object and even the attributes it defines.

We see in the second example that we retrieve the current value of that node attributes, store it in a local variable, and then set up an expectation that the attribute is equal to the String value matcher on the right-hand side.



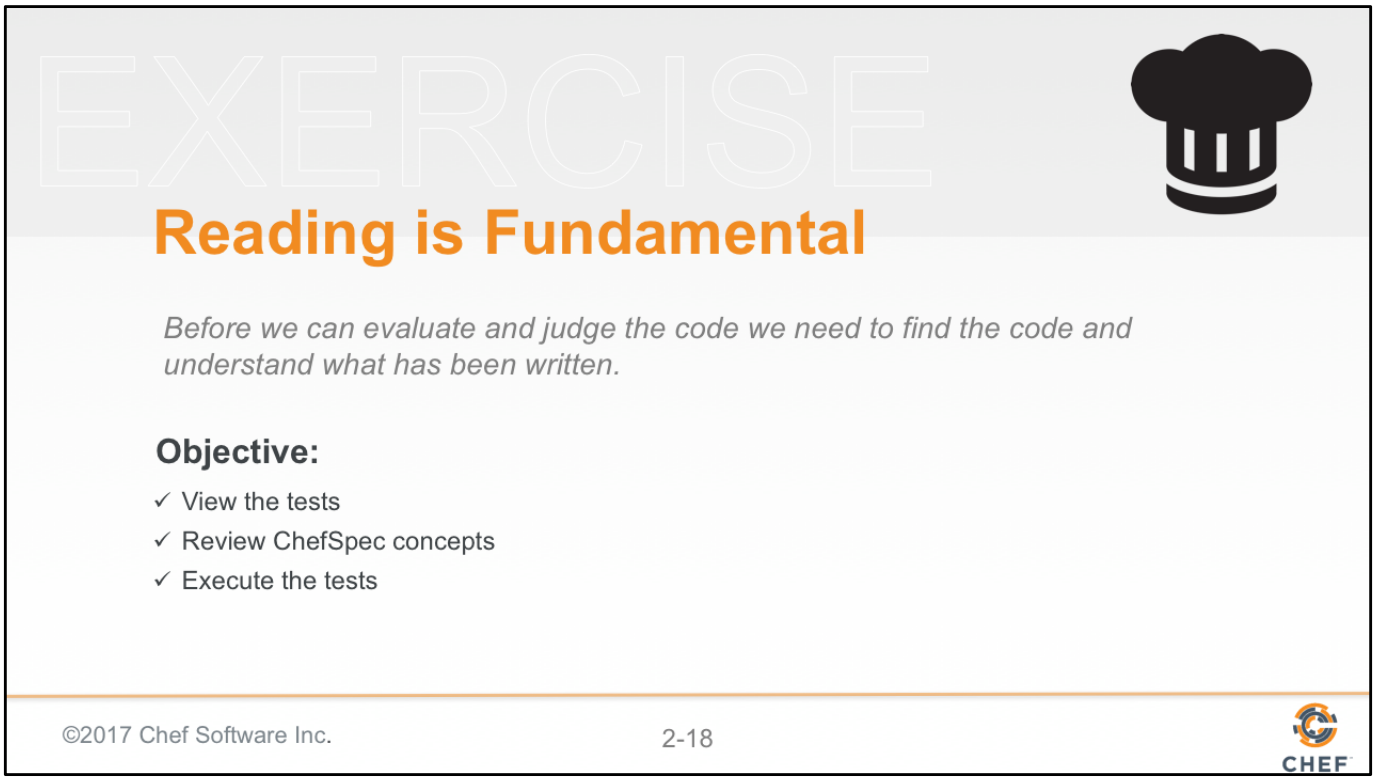
Now that we see the source code and reviewed the important ChefSpec concepts it is time to execute the tests.



Remember ChefSpec is a language that is built on top of RSpec. RSpec provides the command-line application that is able to execute these examples to see their result.

Here we load up the Chef Development Kit (Chef DK) environment and execute the rspec command-line application. We provide a single parameter which is the file that needs to be executed.

Note: If you have the chef tool change correctly configured in your PATH you can simply run 'rspec' and not 'chef exec rspec'. If you have a version of Ruby installed alongside the Chef DK it is often more reliable and safer to use 'chef exec' prefix before running 'rspec'.



We now see all the expectations that are defined within the cookbook pass successfully. This means we can now start to evaluate the code.

EXERCISE

Code Review

- ❑ Form a group
- ❑ Review the code
- ❑ Create a list of feedback

CHEF

For this exercise I want you to form a group, review the code for clarity and purpose, and then create a list of items which the group feels like would help improve the clarity of this code.

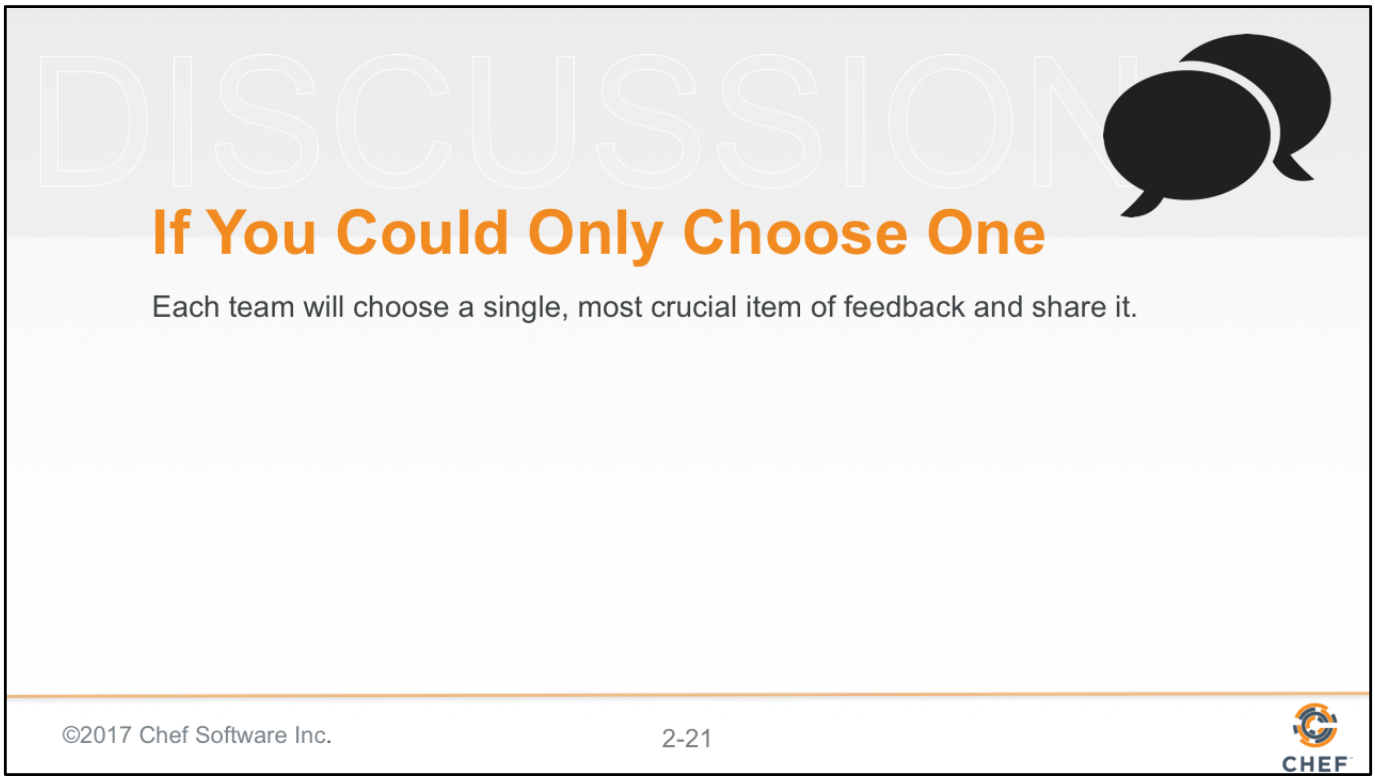
DISCUSSION

Sharing Feedback

Each team will take turns sharing a single item of feedback on the code.

CHEF

Instructor Note: For smaller number of groups this would be a good exercise to do with everyone. For a larger number of groups it may be better to have each group talk with another group or have workshop facilitators walk around and ask each group to share their feedback.



As a group you may have found several different issues. As a group what would you choose if you had to select one thing to address.


CHEF™

The first technique that we are going to explore is the powerful helper called: let.

In this module we will use let to help us express our tests more succinctly.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

CONCEPT

`let`

let allows the author to define a helper method that is available across all the examples within the context it is defined.

Use let to define a memoized helper method. The value will be cached across multiple calls in the same example but not across examples. It is also lazy-evaluated: it is not evaluated until the first time the method it defines is invoked. See `let!` if you want to force the invocation before each example.

https://goo.gl/BJp0lQ

CHEF

When defining our examples and the expectations within them we often find it necessary to setup additional details to support the scenario we are evaluating. These details are often defined inside of a example before the expectation is defined. The same details may be used over-and-over again throughout a specification file as well.

Making a Case for Using the let Helper

```
describe 'ark::default' do
  context 'when no attributes are specifed, on ...platform' do
    it 'installs necessary packages' do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
      expect(chef_run).to install_p...
      expect(chef_run).to install_p...
    end

    it "does not install the gcc-c+...
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
      expect(chef_run).not_to insta...
    end
```


CHEF

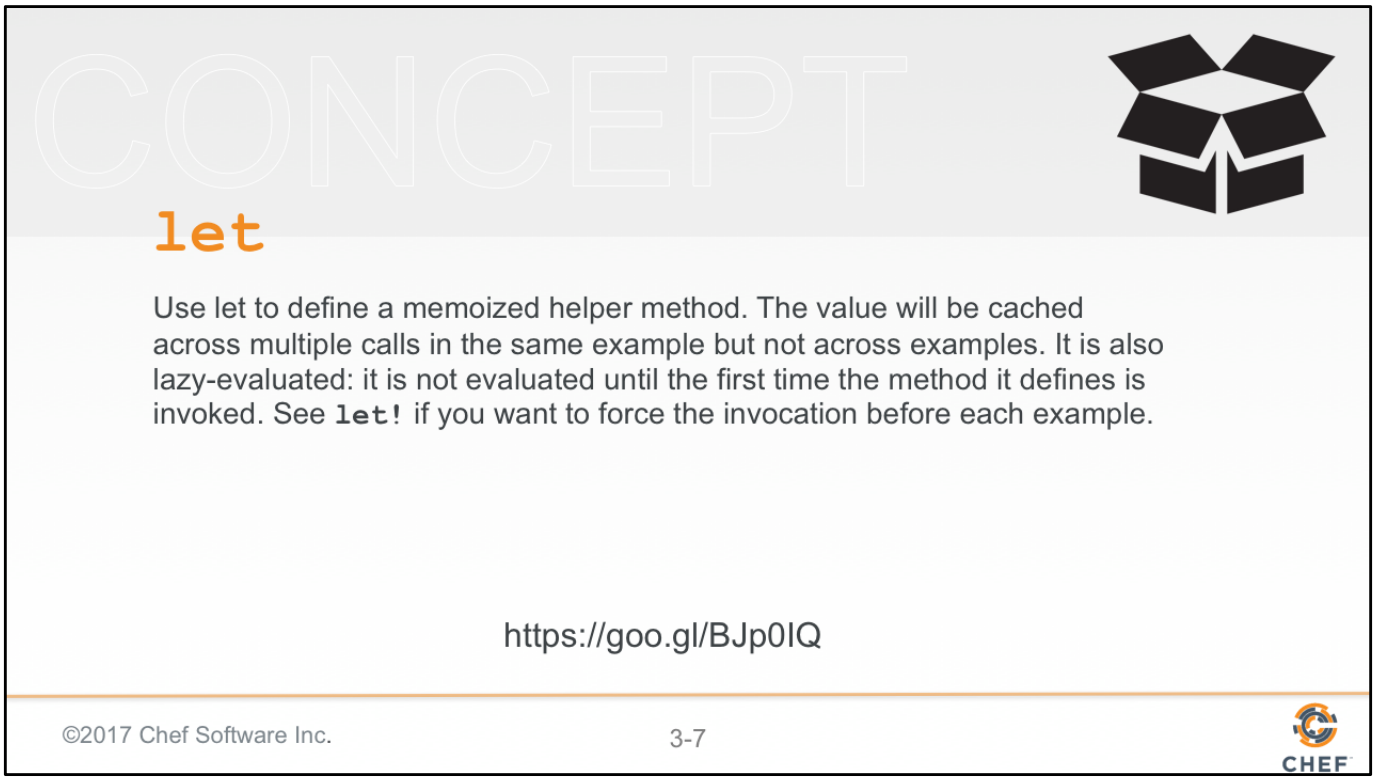
In a specification file you may find yourself redefining particular details over and over again within the setup of each example. Here are two examples that each define a runner object that converges the described recipe. These two lines, to setup the chef_run, is necessary within each example.

Making a Case for Using the let Helper

```
describe 'ark::default' do
  context 'when no attributes are specifed, on ...platform' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
    end

    it 'installs necessary packages' do
      expect(chef_run).to install_p...
      expect(chef_run).to install_p...
    end

    it "does not install the gcc-c+...
      expect(chef_run).not_to insta...
    end
```


Here with the use of let we are able to extract that setup information into a single location that does a good describing what it means when we use chef_run within each example within this context.

A helper defined in a let block takes a symbol parameter which defines the helper method that is later invoked within the examples. The block of code (between the do and the end) contains the code that is executed. The result of last line within the block is returned when it is used within an example.



let defines a memoized helper. Memoization is an optimization technique that means once it is evaluated the result is stored and subsequent requests during the same example return the previously requested value.

You could think of it as a cache-miss, the result is retrieved and stored. Then all subsequent requests, within the example, are cache-hits.

The let helper is often called lazy; it is not evaluated until it is first requested. Again an optimization technique.

Note: If you ever want the code evaluated before it's first use take a look at **let!**

Diagramming the let Helper Method

```
describe 'ark::default' do
  context when no attributes are ...
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
      runner
    end

    it 'installs necessary packages' do
      expect(chef_run).to install_p...
      expect(chef_run).to install_p...
    end

    it "does not install the gcc-c+...
      expect(chef_run).not_to insta...
    end
```

1. let is a RSpec helper method
2. Ruby Symbol
3. Code Block
4. Invocation

Invoking the Helper Method

```
describe 'ark::default' do
  context 'when no attributes are ...
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
      runner
    end

    it 'installs necessary packages' do
      expect(chef_run).to install_p...
      expect(chef_run).to install_p...
    end

    it "does not install the gcc-c+...
      expect(chef_run).not_to insta...
    end
```

1. chef_run sends a message
2. RSpec invokes the contents of the block
3. RSpec stores the contents of the execution
4. chef_run sends a message
5. RSpec retrieves the stored execution

Here we see the invocation of the chef_run 1, which evaluates the contents of the let helper block 2. The result of the block is stored or cached and then returned to this specific example 3. The next usage of chef_run 4 will retrieve the stored evaluation 5 instead of invoking the code within the let helper again.

Cached Within each Example

```
describe 'ark::default' do
  context 'when no attributes are ...
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
      runner
    end

    it 'installs necessary packages' do
      expect(chef_run).to install_p...
      expect(chef_run).to install_p...
    end

    it "does not install the gcc-c+...
      expect(chef_run).not_to insta...
    end
```

1. chef_run is loaded and stored
2. chef_run uses the stored invocation
3. chef_run is loaded and stored

Here we see again in the first example we invoke the let helper for the first time 1. When we use chef_run again in the same example 2 the same memoized or cached value is used.

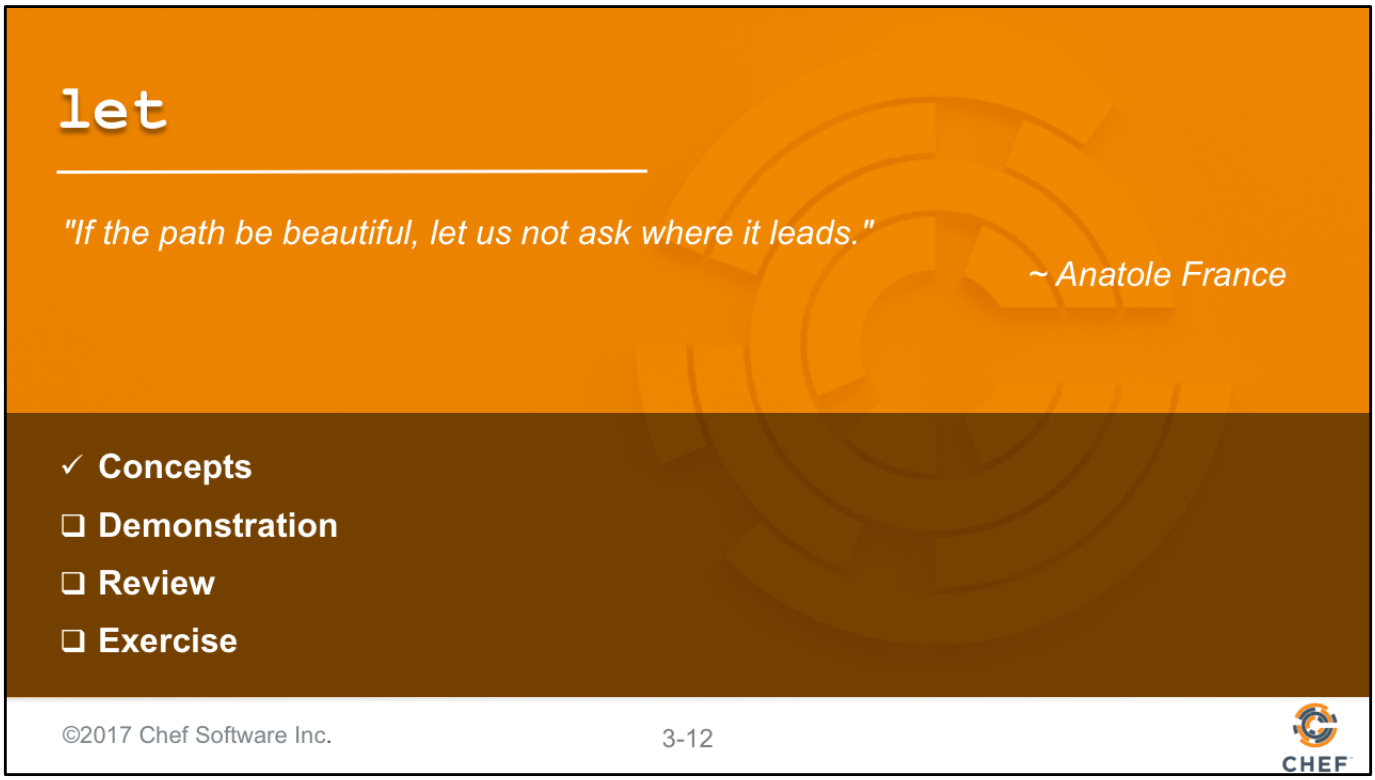
Within the next example chef_run is used 3. Because this use exists within a different example the let helper is again invoked and the value is stored.

This is an important feature as it ensures that between each example is independent of each other. Ensuring that any choices we made above to create the scenario above does not effect this new run.

A chef_run with Node Attributes

```
describe 'ark::default' do
  context 'when no attributes are specified, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new({ platform: 'centos',
                                          version: '6.7' })
      runner.converge(described_recipe)
    end

    # ... EXAMPLES WITHIN CONTEXT
  end
end
```


Use of let is already in the specification file we are examining and is automatically generated with each specification file.



Now it is time to demonstrate how let can be used to help create more succinct examples.

DEMO

Live Demonstration

https://goo.gl/ChkP47

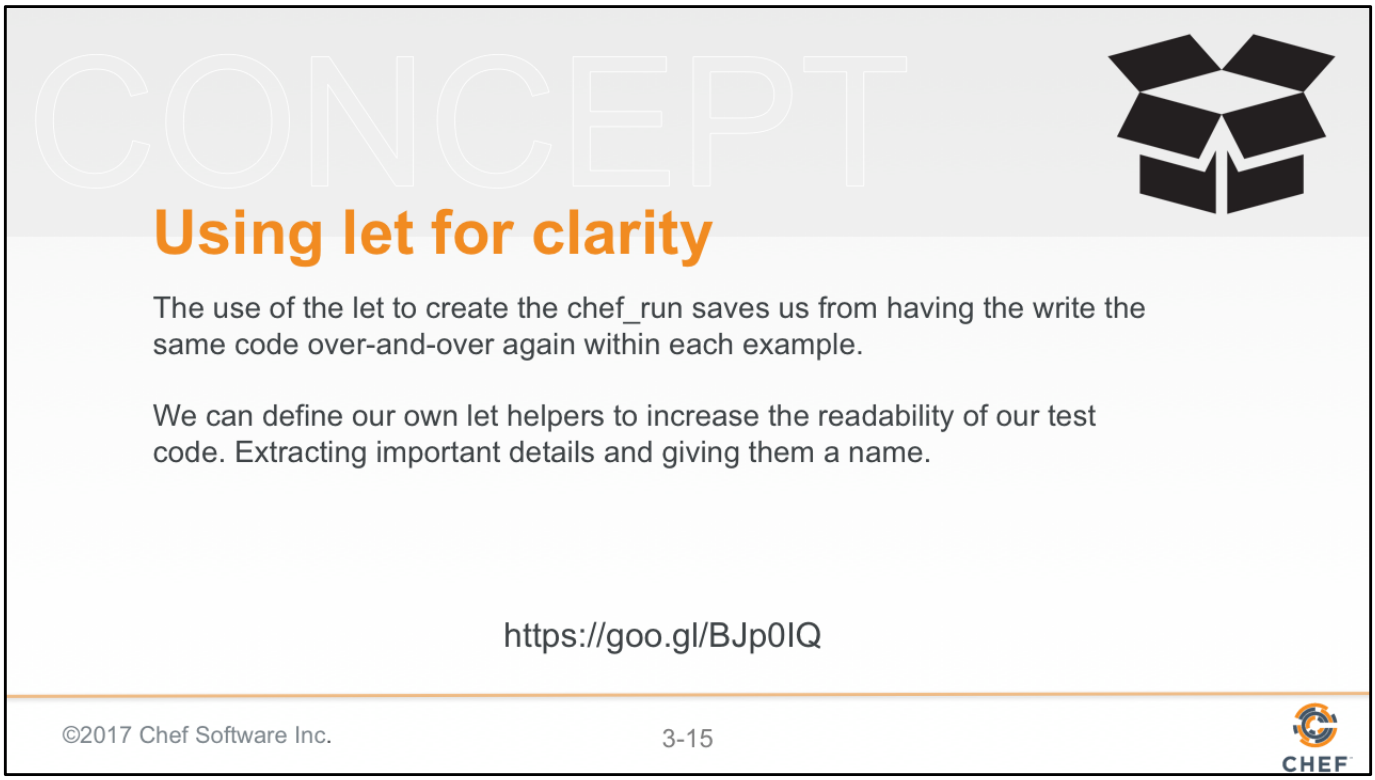
CHEF

let

"Let us always meet each other with smile, for the smile is the beginning of love."
~ Mother Teresa

- ✓ **Concepts**
- ✓ **Demonstration**
- ❑ **Review**
- ❑ **Exercise**

CHEF



As you saw in the demonstration we can use let for clarity. It is a useful strategy to increase readability and giving a name to particular set of data.

A chef_run with Node Attributes

```
describe 'ark::default' do
  context 'when no attributes are specified, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new({ platform: 'centos',
                                          version: '6.7' })
      runner.converge(described_recipe)
    end

    # ... EXAMPLES WITHIN CONTEXT
  end
end
```


Here again is the chef_run which defines a chef run with a particular set of platform details. A hash of node attributes are provided to the runner to help simulate that particular platform.

Using let to Create Clearer Examples

```
describe 'ark::default' do
  context 'when no attributes are specified, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new(node_attributes)
      runner.converge(described_recipe)
    end

    let(:node_attributes) do
      { platform: 'centos', version: '6.7' }
    end

    # ... EXAMPLES WITHIN CONTEXT
  end
```


CHEF

Using a let allows us to name these details, making it easier to see the specifics of the platform we are examining.

Within the block of a let helper you can use other let helpers. When the chef_run helper is invoked it will in turn invoke the node_attributes helper.

Note: The order in which you define these let helpers is not important as they are lazy evaluated.

Overriding a Helper in a Child Example Group

```
describe 'ark::default' do
  let(:chef_run) do
    runner = ChefSpec::SoloRunner.new(node_attributes)
    runner.converge(described_recipe)
  end

  let(:node_attributes) do
    {}   # This is an empty hash, the default to be overridden
  end
  context 'when no attributes are specified, on CentOS' do
    let(:node_attributes) do
      { platform: 'centos', version: '6.7' }
    end
    # ... EXAMPLES WITHIN CONTEXT
  end
```


Remember a let helper is available within any example group that it is defined. Adding a let helper to a parent example group makes it available in any child context. This means you can define it once in the parent example group. If the helper defined in the parent example group does not suit the needs of the child context you are always able to redefine it.

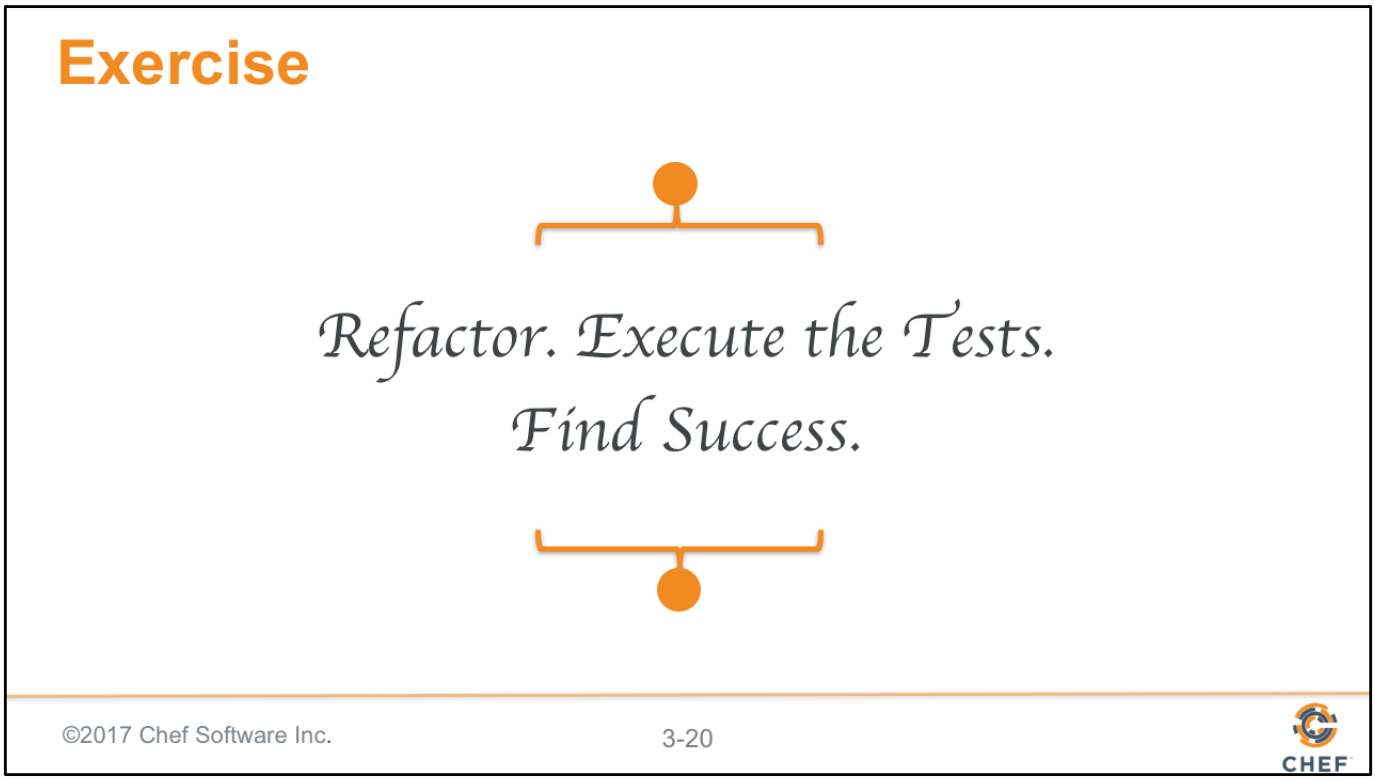
This is a great example of how to extract the common chef_run helper, state it once in the top-most example group and then override the particular node attributes in each child example group.

let

"Let us not pray to be sheltered from dangers but to be fearless when facing them."
~ Rabindranath Tagore

- ✓ **Concepts**
- ✓ **Demonstration**
- ✓ **Review**
- ❑ **Exercise**

CHEF



Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

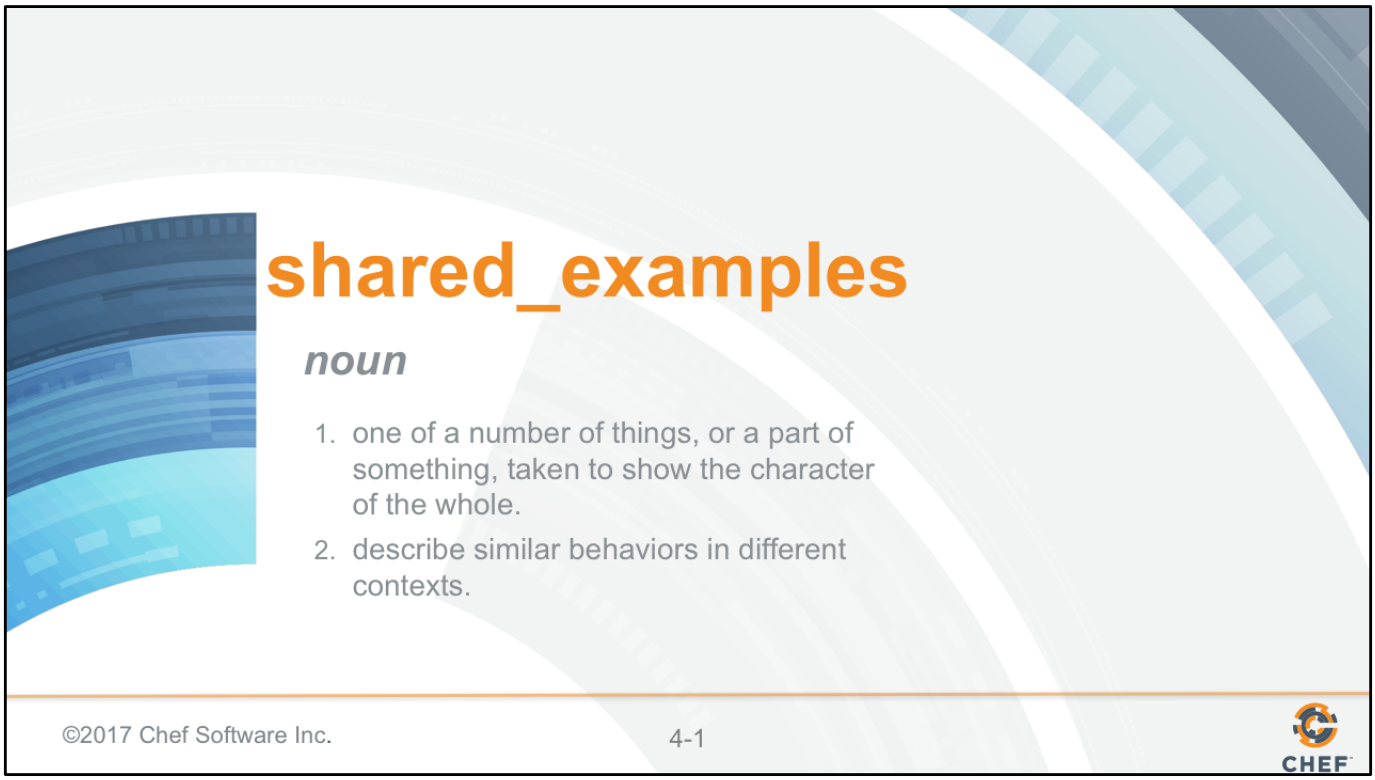
let

"Come, let us have some tea and continue to talk about happy things."
~ Chaim Potok

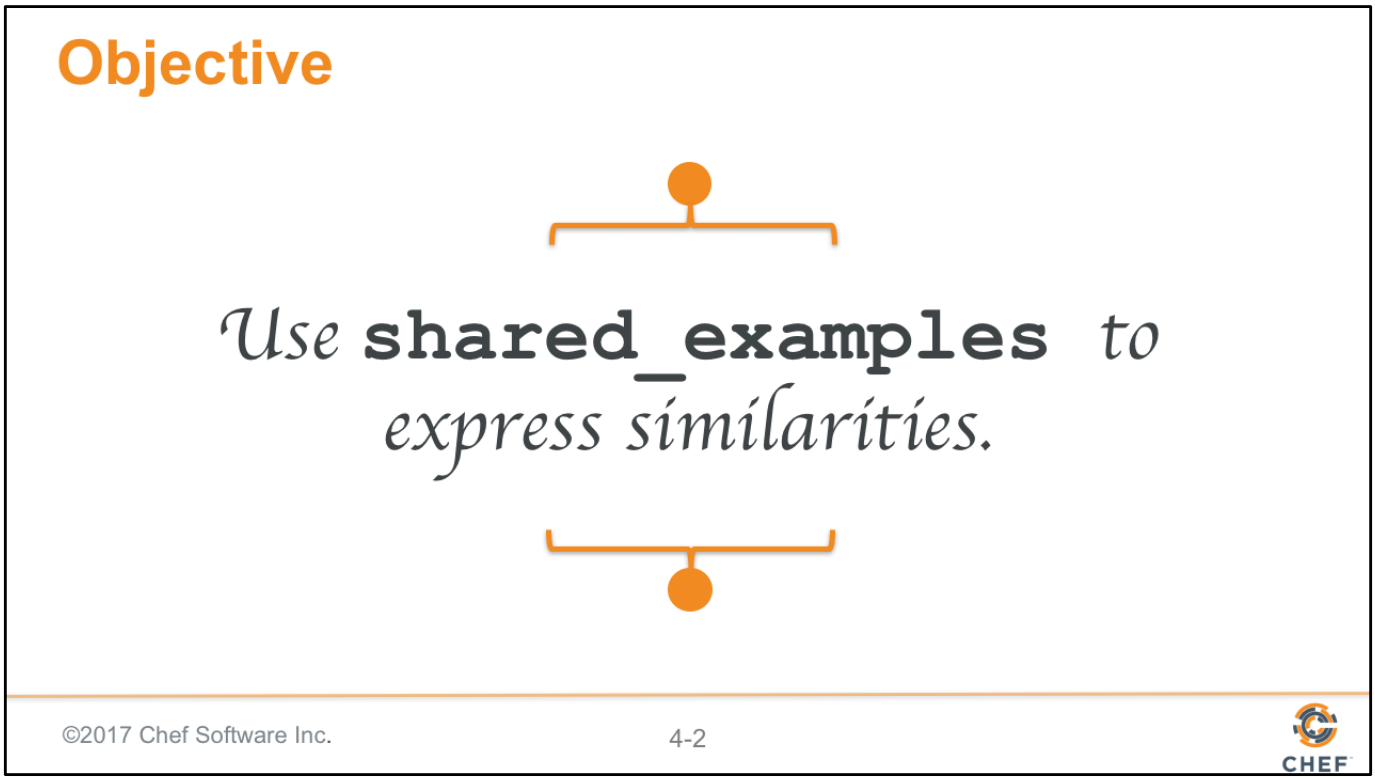
- ✓ **Concepts**
- ✓ **Demonstration**
- ✓ **Review**
- ✓ **Exercise**

CHEF


CHEF™



The second technique we are going to explore is shared_examples.



The objective of this module is to use shared_examples to help us express our similar examples once in one location and reference those examples.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

CONCEPT

shared_examples

Shared examples let you describe behavior of classes or modules. When declared, a shared group's content is stored. It is only realized in the context of another example group, which provides any context the shared group needs to run.

https://goo.gl/yi12tM

CHEF

When we write our examples we may find ourselves again repeating ourselves or at least it may feel that way. There are ways in RSpec that allows us to extract the examples that we see that are common and re-use them between each of our different example groups.

This can be done through RSpec's shared_examples.

Viewing an Example Specific to a Platform

```
context 'when no attributes are specified, on an unspecif...orm' do
  it 'installs necessary packages' do
    expect(chef_run).to install_package('libtool')
    expect(chef_run).to install_package('autoconf')
    expect(chef_run).to install_package('unzip')
    expect(chef_run).to install_package('rsync')
    expect(chef_run).to install_package('make')
    expect(chef_run).to install_package('gcc')
    expect(chef_run).to install_package('autogen')
  end

  # ... remainder of examples within defined within this example group
end
```


Within this example we have repeated ourselves a number of times ensuring that the packages are all installed on the unspecified platform.

Note: Defining multiple expectations this way can cause problems when executed. Imagine that the recipe as written did not install the first package or the last package. When executing the tests RSpec would report that the first expectation failed but never run any of the remaining expectations. Hiding a failing expectation.

Some people that write tests argue that it is best to write one expectation per example. That way you would never hide any failing expectations.

However, creating an example per expectation increases the length of the test execution which may prevent you from executing the examples in the first place. It also makes it a little more work to maintain and adding more code creates more cognitive load on future maintainers.

Refactoring the Example

```
context 'when no attributes are specified, on an unspecif...orm' do
  it 'installs necessary packages' do
    installed_packages = %w[ libtool autoconf unzip rsync make gcc autogen ]
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end

  # ... remainder of examples within defined within this example group
end
```


We can start by making this example more elegant by defining an array that contains all the package names. Then iterating through each name in a loop and making an assertion about each package being installed.

Note: Traditionally arrays are defined with unadorned square brackets. Each element within the array is separated by commas. Strings, which these package names are, would need to have quotes around them. The above array could also be written as the following:

```
installed_packages = [ 'libtool', 'autoconf', 'unzip', 'rsync',
'make', 'gcc', 'autogen' ]
```

The `%w`, used above, states the contents of the Array are Strings and the whitespace between each element serves as the delimiter. This makes it far easier to maintain Arrays of Strings.

Extracting the Packages from the Example

```
context 'when no attributes are specified, on an unspecif...orm' do
  let(:installed_packages) do
    %w[ libtool autoconf unzip rsync make gcc autogen ]
  end

  it 'installs the necessary packages' do
    installed_packages = %w[ libtool autoconf unzip rsync make gcc autogen ]
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
```

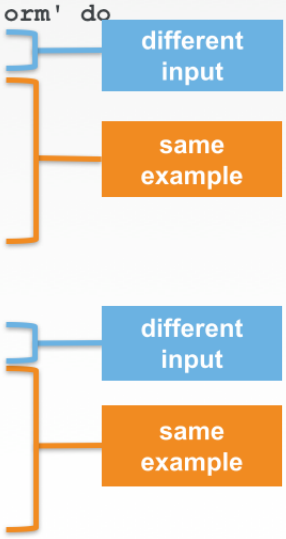

CHEF

The array defined in the example can be extracted into a let example. Similar to how we extracted the node attributes used within the chef_run helper we defined.

This should hopefully make it clearer what packages are being installed on the system and make it far easier to understand and update this list if there were changes or issues discovered later.

Finding Similar Expressed Expectations

```
context 'when no attributes are specified, on an unspecif...orm' do
  let(:installed_packages) ...
  it 'installs the necessary packages' do
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
context 'when no attributes are specified, on CentOS' do
  let(:installed_packages) ...
  it 'installs the necessary packages' do
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
```



CHEF

If you were to perform this same refactoring technique for each of the 'installs the necessary packages' examples you would find that at the end of it all you would have a lot of similar looking examples. The element that would change in each example group is the Array of package names.

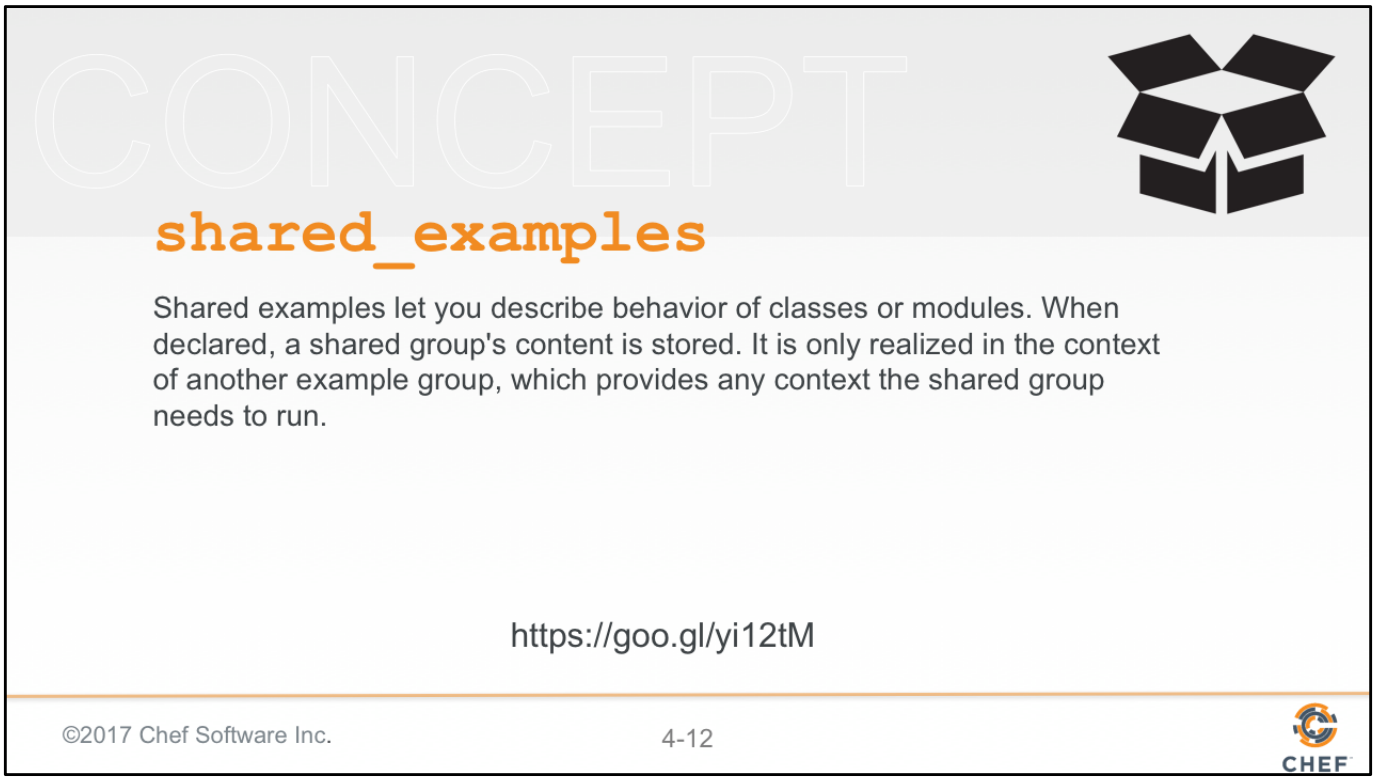
Note: This technique reminds me of the same techniques when working in the field of math. Before adding fractions together you often need to find the common denominator. To solve for a unknown value in Algebra you often found ways to restructure the equation so as to apply a technique. That is exactly what is happening here.

Now it is time to demonstrate how shared_exaples can be used to help create more elegant example groups.

In this demonstration I will show you how to extract the package names into a let helper, create similar looking examples, and then extract those examples into a shared_examples block.

shared_examples

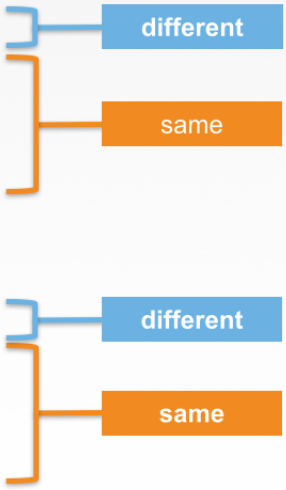
- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise



shared_examples allow us to describe examples independent of an example group and then include those examples within the example groups when needed.

Finding Similar Expressed Expectations

```
context 'when no attributes are specified, on an unspecif...orm' do
  let(:installed_packages) ...
  it 'installs the necessary packages' do
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
context 'when no attributes are specified, on CentOS' do
  let(:installed_packages) ...
  it 'installs the necessary packages' do
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
```



I refactored the code to look like the following across at least two example groups to create identical examples.

A Place to Share Examples

```
shared_examples 'installs packages' do
  it 'installs the necessary packages' do
    installed_packages.each do |name|
      expect(chef_run).to install_package(name)
    end
  end
end
context 'when no attributes are specified, on an unspecif...orm' do
  let(:installed_packages) ...
  it_behaves_like 'installs packages'
end
context 'when no attributes are specified, on CentOS' do
  let(:installed_packages) ...
  it_behaves_like 'installs packages'
end
```


We can then take that example and move it to its own special context which is defined with a special block named shared_examples.

The shared_examples method takes an argument which is the name that describes all the examples that reside within it. Defining the example is the same as you would define examples in an example group.

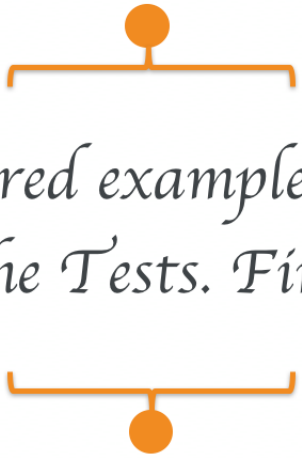
The examples defined within shared_examples is not evaluated until it is invoked within an example group through the method 'it_behaves_like'. This method accepts a single parameter and that is the named of the shared_examples group we defined above.

Note: The shared_examples helper must be defined before its use in the first example group. So often you can move it to the top of the specification file or within the parent context.

shared_examples

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise

Exercise



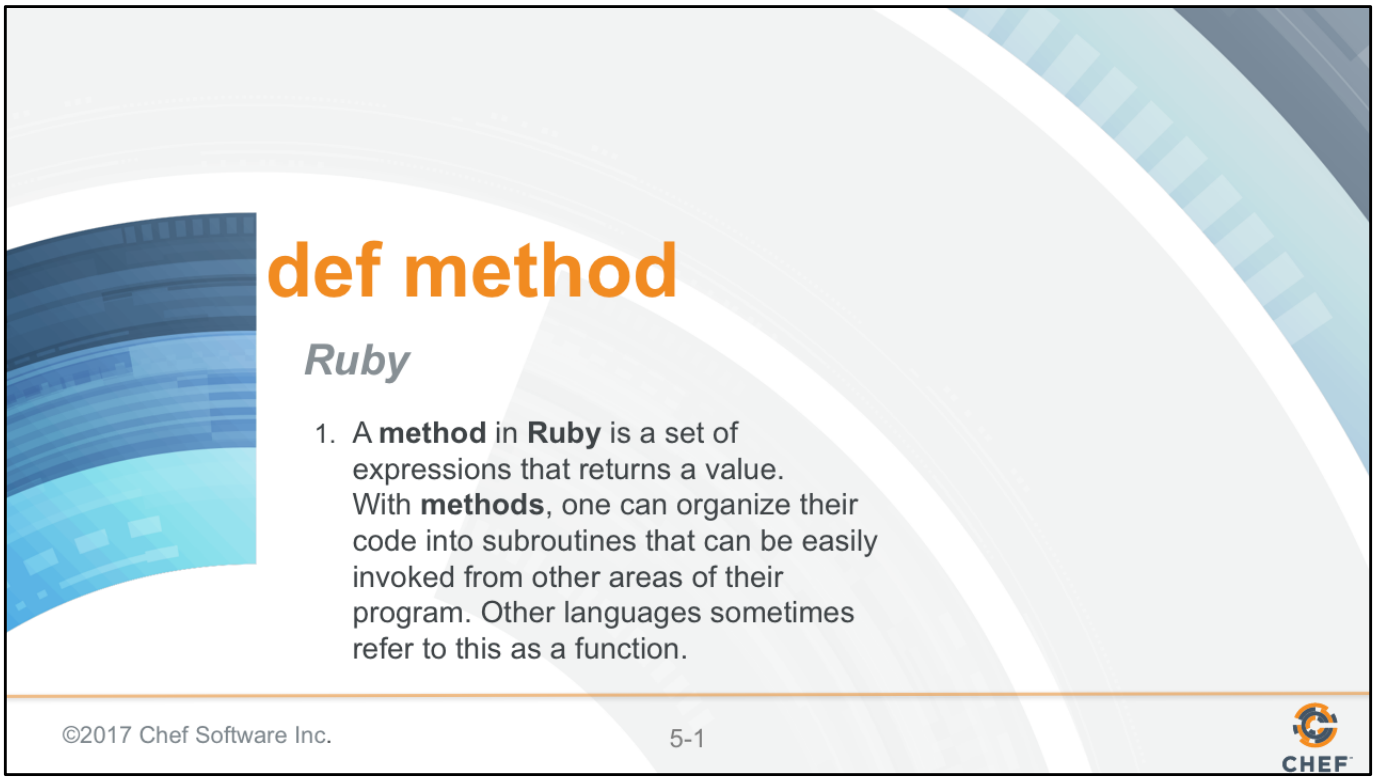
Find shared examples; extract.
Execute the Tests. Find Success.

Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

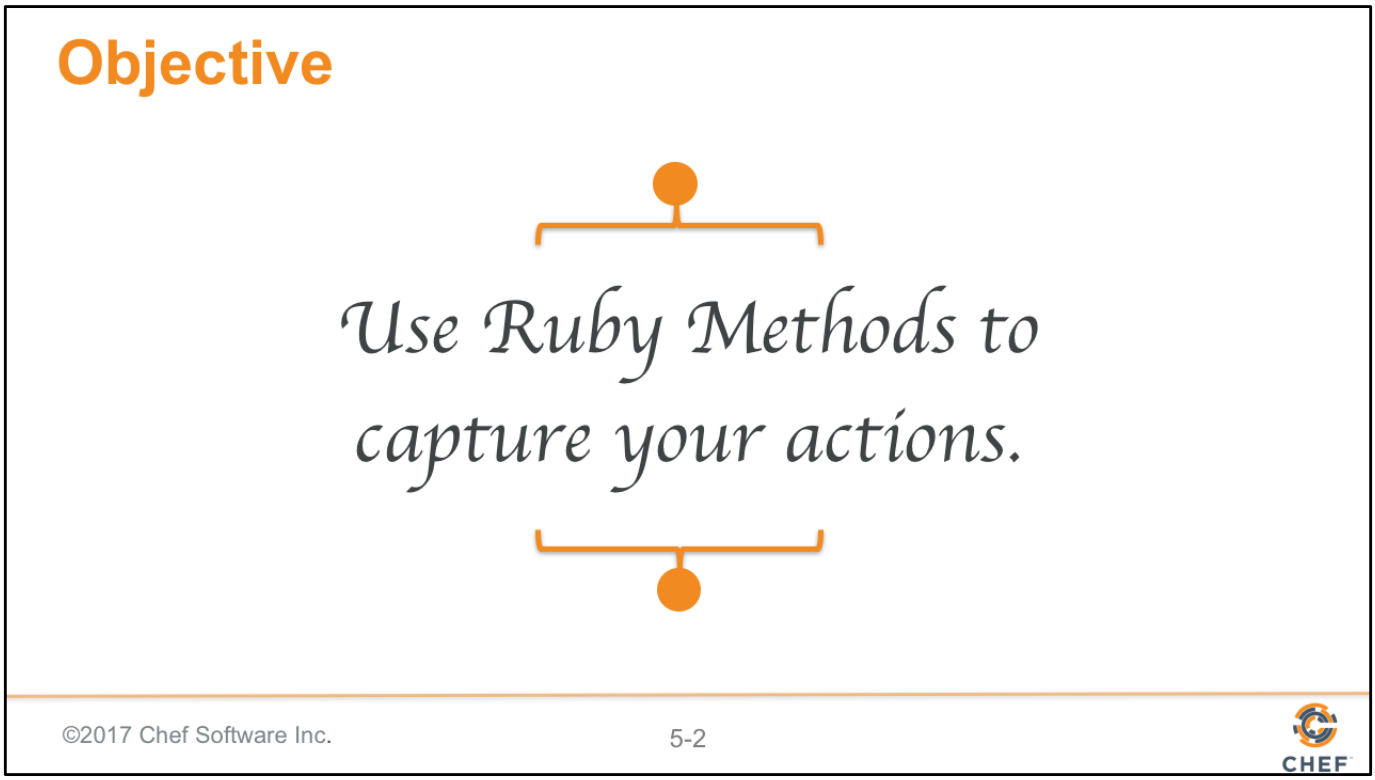
shared_examples

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise


CHEF™



The third technique we are going to explore is defining Ruby methods.



The objective of this module is to use ruby methods to help us capture operations we repeatedly perform through our specification file.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

CONCEPT

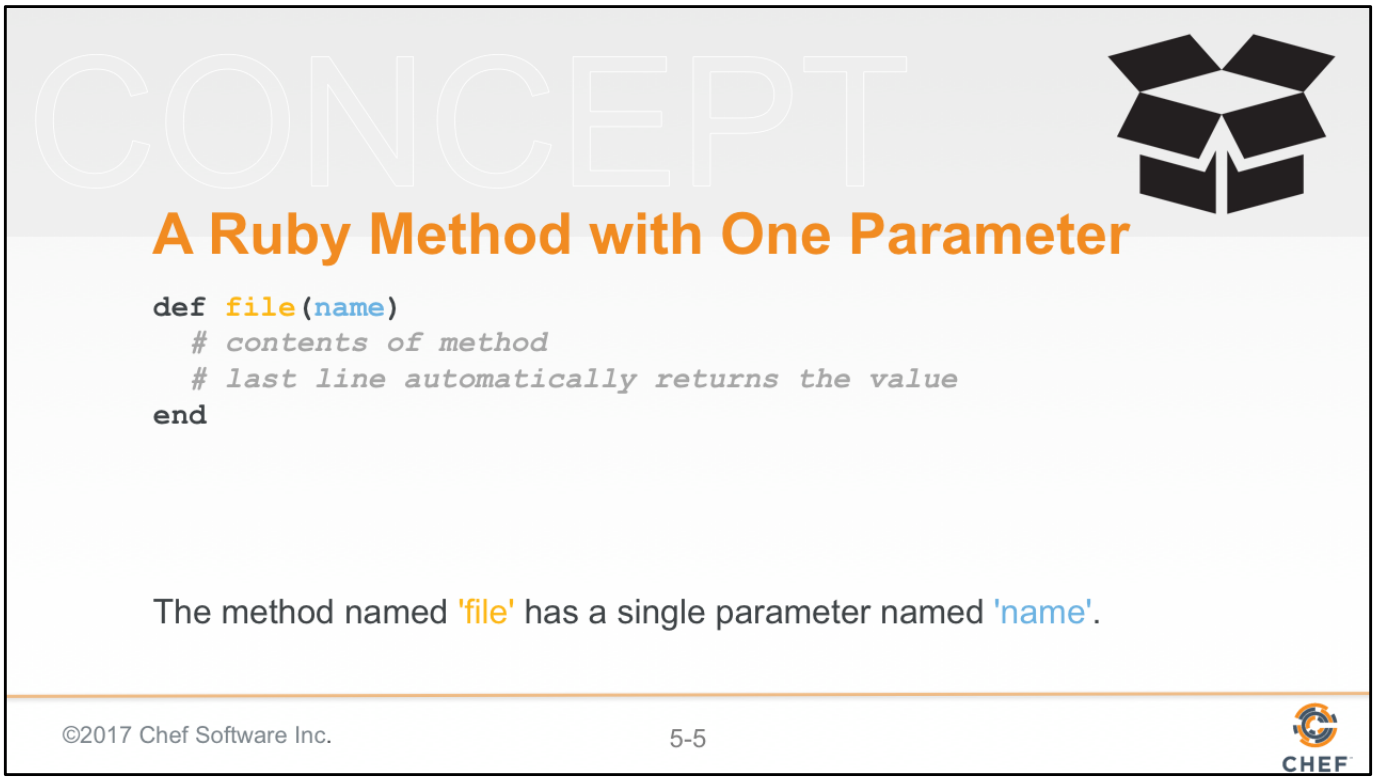
Create helper method

You can define a Ruby method that takes care of some of the tedious work of retrieving node attributes.

CHEF

We have seen how the let helper allows us to define common information in a central location and then re-use that information throughout our examples and in our expectations.

However, there are some limits to the let helpers. Namely the only return a static information. They cannot repeat a process that takes inputs. That is where Ruby methods can help us.

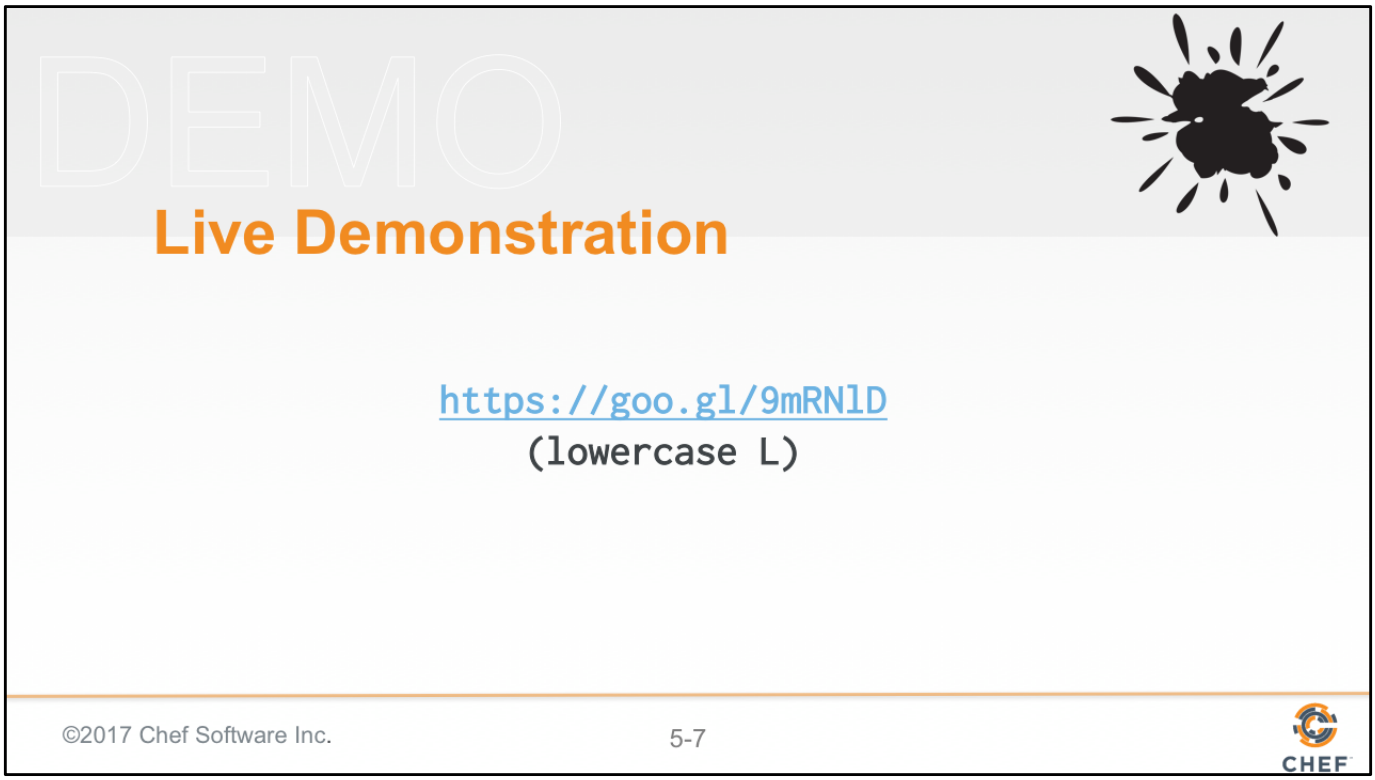


A ruby method starts with the Ruby keyword def, is followed by the name of the method, and then we define a list of parameters.

Within a method you can define a series of expressions and statements that perform an operation. The last line of a Ruby method will automatically be returned as a result of the method when it is invoked.

Note: Ruby methods can also accept no parameters, named paramaters, an unbounded set of parameters, and blocks. This is far beyond the scope of this workshop to describe all the ways in which methods can be defined and used.

Now it is time to demonstrate how defining a method can be used to help create more elegance within each example.



In this demonstration I will show you how to define a method that accepts a single parameter, an attribute name, and then return the node attribute from the node object contained within the chef_run.

def method

- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise

Viewing the Repetition of Retrieving Attributes

```
it "apache mirror" do
  attribute = chef_run.node['ark']['apache_mirror']
  expect(attribute).to eq "http://apache.mirrors.tds.net"
end

it "prefix root" do
  attribute = chef_run.node['ark']['prefix_root']
  expect(attribute).to eq "/usr/local"
end
```


CHEF

Within the specification we had a number of examples that setup expectations around ensuring that the correct node attributes were set based on the platform. If you review the attributes file within the cookbook you can see why this kind of information would be useful as there is a lot of conditional logic used to determine the correct values.

Refactoring with let to help ease the pain

```
let(:node) do
  chef_run.node
end

it "apache mirror" do
  attribute = node['ark']['apache_mirror']
  expect(attribute).to eq "http://apache.mirrors.tds.net"
end

it "prefix root" do
  attribute = node['ark']['prefix_root']
  expect(attribute).to eq "/usr/local"
end
```


CHEF

We can use a let helper make it easier to gain access to the node object. However, when it comes to each individual node attribute we would not be able to access each one through a let helper unless we created a let helper for each attribute. This is not ideal and more likely would create more confusion when reading our specification.

Implementing a Helper Method

```
let(:node) do
  chef_run.node
end

def attribute(name)
  node[described_cookbook][name]
end
it "apache mirror" do
  expect(attribute('apache_mirror')).to eq "http://apache.mirr....net"
end

it "prefix root" do
  expect(attribute('prefix_root')).to eq "/usr/local"
end
```


This is a perfect moment for us to use a method. We define a method named 'attribute' which accepts one parameter, the name of the attribute defined within the cookbook. Within the body of the method we use the node object returned by the let helper defined above. We use a method named 'described_cookbook' here that ChefSpec creates for us based on the 'described_recipe' defined at the beginning of the specification.

Note: The method can be defined anywhere within the specification, as long as you only use the method within the examples. Though I would suggest moving this to the top of the file to make it clear what are helpers and what are specifications.

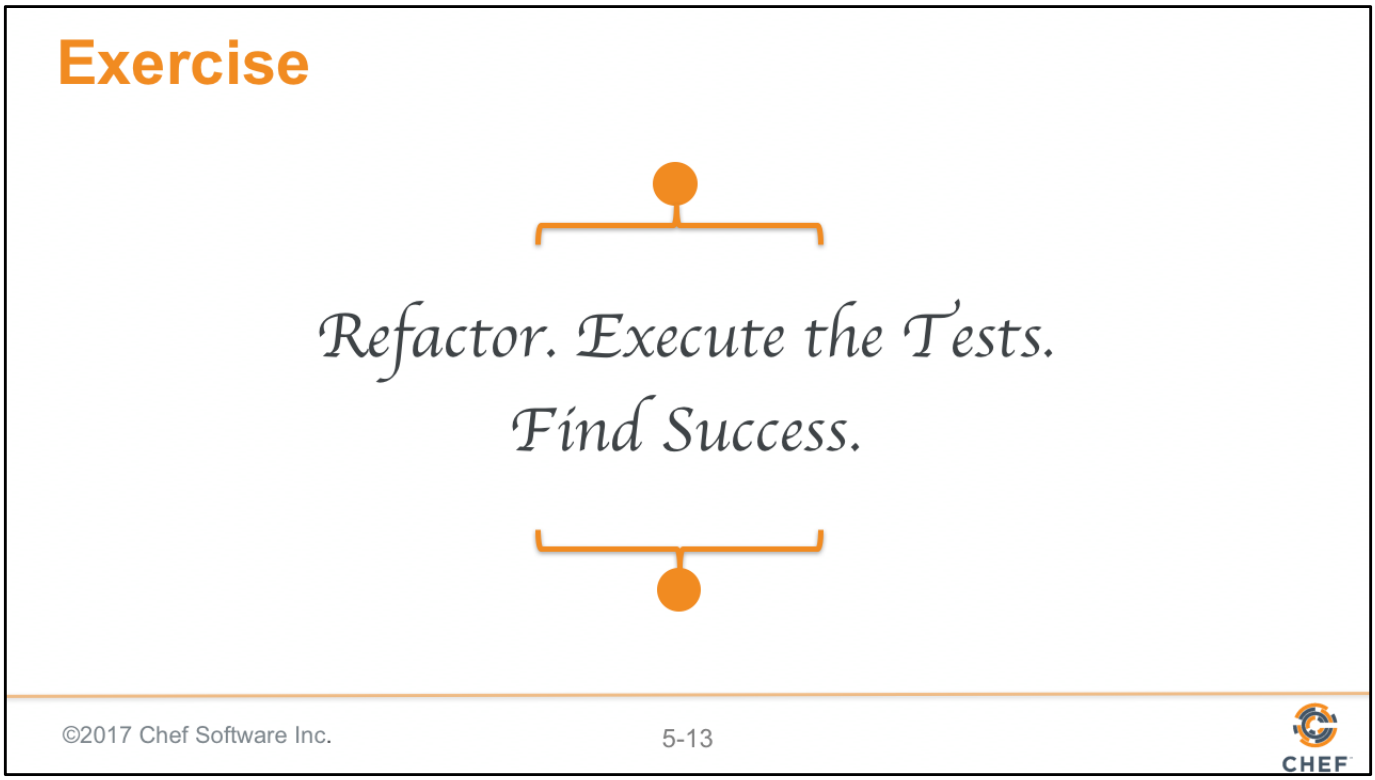
Note: The name of the method 'attribute' may be misleading or confusing for some people. 'cookbook_attribute' may be more accurate as you are defining a method that is checking for an attribute defined. However, it is probably rare that you would testing to ensure attributes defined in a different cookbook were being set. Ultimately, the name of the method is up to you and your team. Either 'attribute' or 'cookbook_attribute' feel appropriate and create expectations that read well to future maintainers.

Note: Instead of using 'described_recipe' we could have used the name of the cookbook. Using 'described_recipe' allows for this helper method to be more portable to other cookbooks.

def method

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise

CHEF

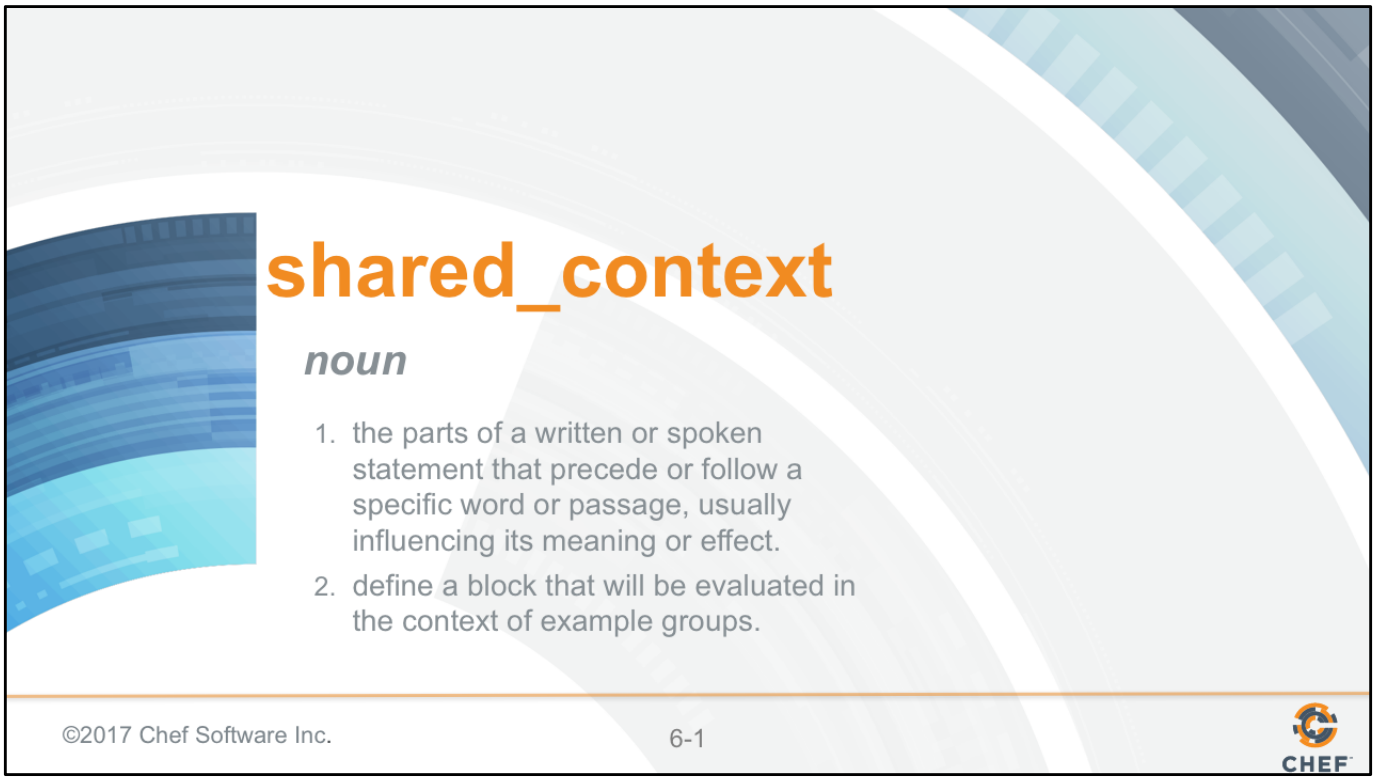


Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

def method

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise


CHEF™



The fourth technique we are going to explore is shared_context.

Objective

In this module we will use shared_context to capture all our helpers in a single place; saving all we wrote to use another day.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

CONCEPT

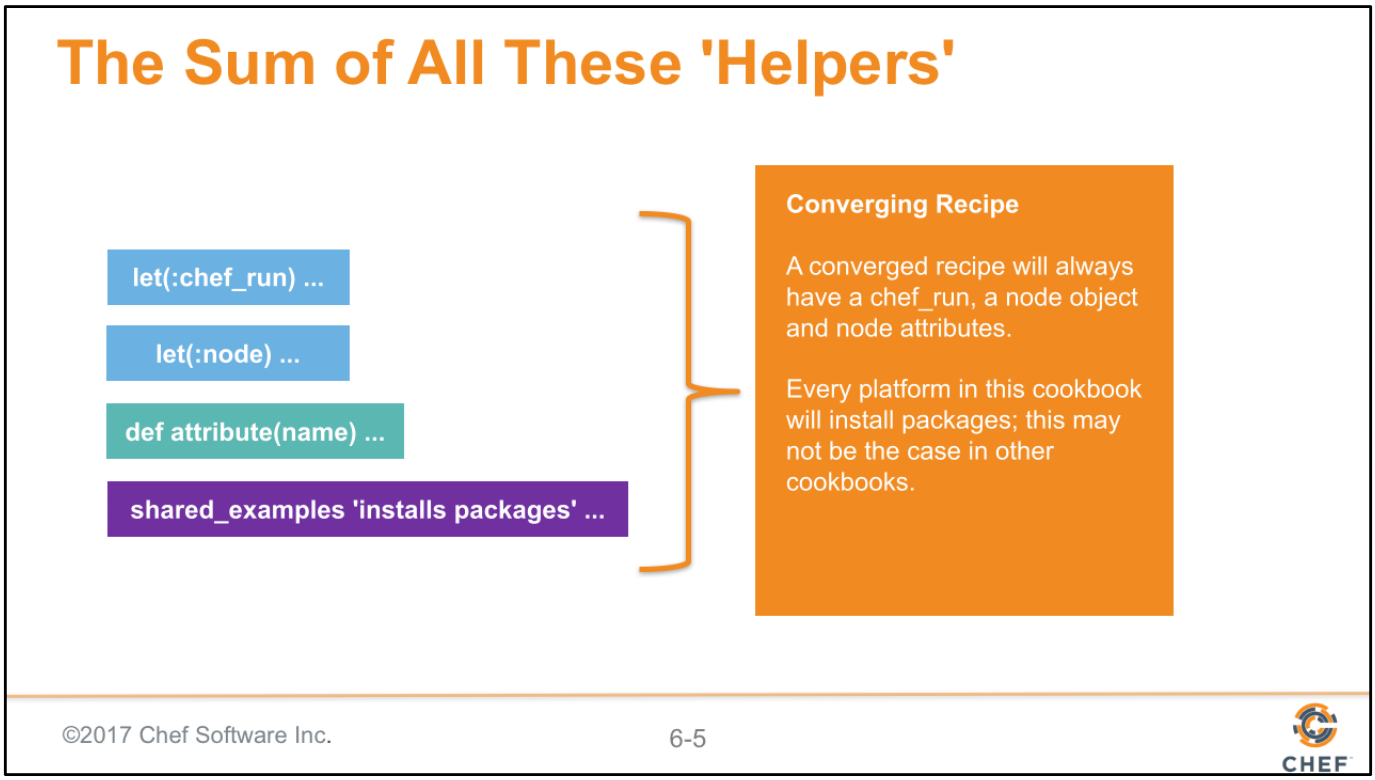
shared_context

Use **shared_context** to define a block that will be evaluated in the context of example groups either explicitly, using **include_context**, or implicitly by matching metadata.

http://goo.gl/R0ujTA

CHEF

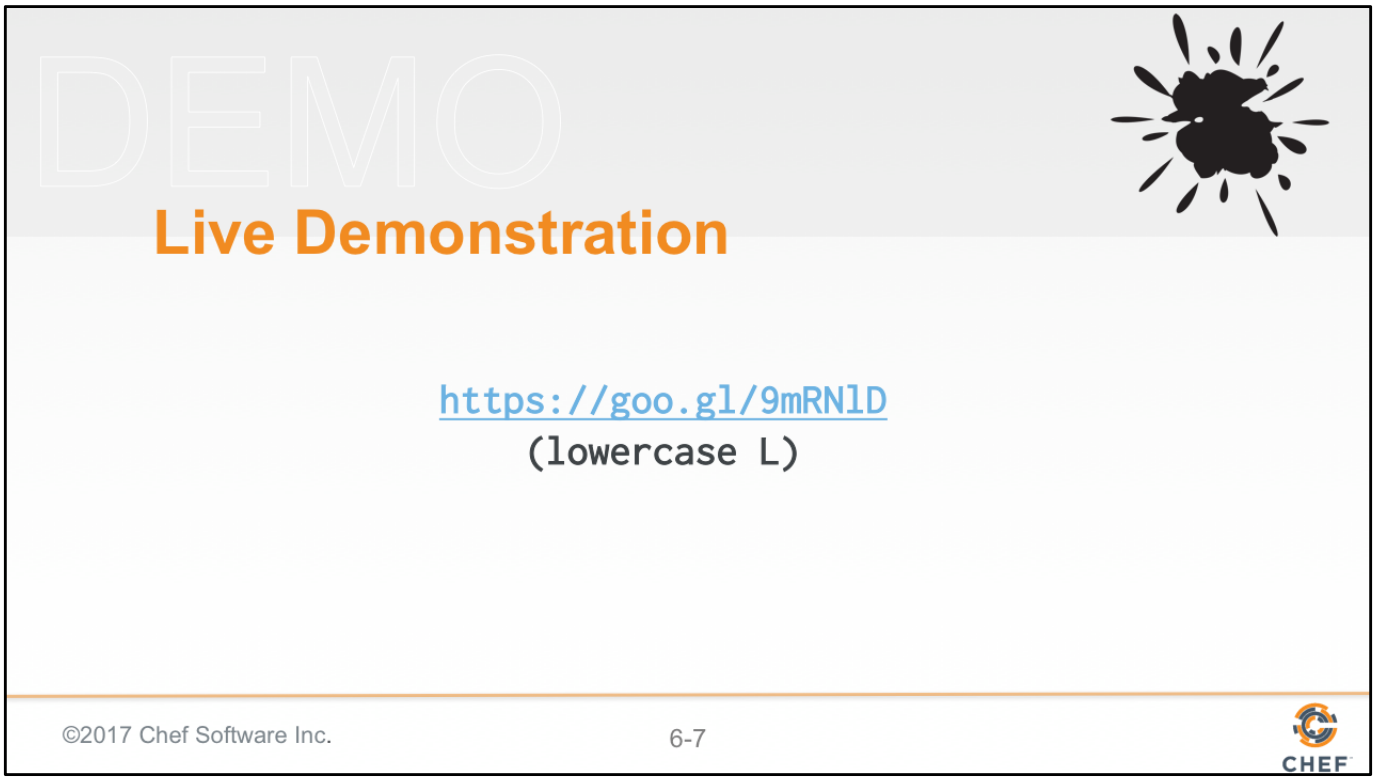
We have defined helpers with common helpers with let, examples that we have shared, and methods to assist us to retrieve node attributes. All of these these exist as separate helpers. A shared_context is a way for us to collect all these useful tools in a single place, give them a name, and then include them anywhere they are needed.



When we examine all of the helpers that we have created so far to help us make our specifications clearer we see that they are are all attempting to describe when a recipe converges on a system.

A converged system will always have a chef_run and a node object. You will also likely want to access node attributes. You could make an argument that installing packages does not take place in every recipe and not make it part of this shared context 'converging recipes'. With this cookbook the default recipe and all platforms definitely install some packages but that is not the case for every cookbook. As always correctly naming these things that you create is difficult to do correctly.

Now it is time to demonstrate how shared_context can be used to collect up all these helpers into a single context that we can apply to example group and possible future example groups that we create.



In this demonstration I will show you how to define a shared_context, move the existing helpers into this context, and then apply this context to a example group.

shared_context

- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise

Repeating More than Examples

```
describe 'ark::default' do
  context 'when no attributes are specified, on an uns...platform' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new(node_attributes)
      runner.converge(described_recipe)
    end

    let(:node_attributes) do
      {}
    end

    def attribute(name) ...
```


CHEF

As we saw we were creating all these let helpers, methods and shared examples all are focused on the goal of describing a converged recipe. If were to starting thinking about re-using this functionality in other places it would be good to move all these helpers into a single context that we can import wherever we needed it.

Repeating More than Examples

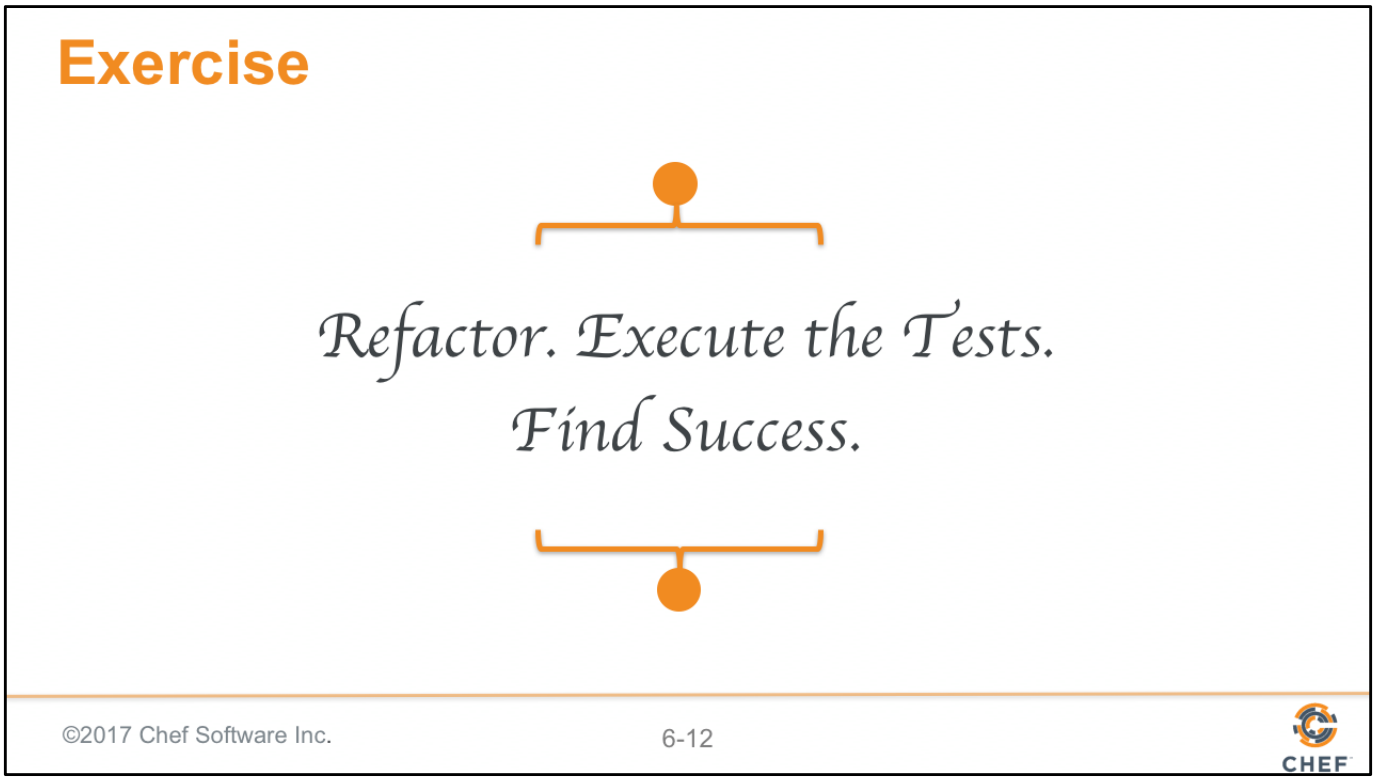
~/ark/spec/unit/recipes/default_spec.rb

```
require 'spec_helper'

shared_context 'converged recipe' do
 let(:chef_run) do
   runner = ChefSpec::SoloRunner.new(node_attributes)
   runner.converge(described_recipe)
 end

  let(:node_attributes) do
    {}
  end

  def attribute(name) ...
end

describe 'ark::default' do
  context 'when no attributes are specified, on an uns...platform' do
    include_context 'converged recipe'
```


We create the shared_context, give it a name, and then move any let helpers, methods we've defined and even shared_examples that we've defined and included.

shared_context

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise

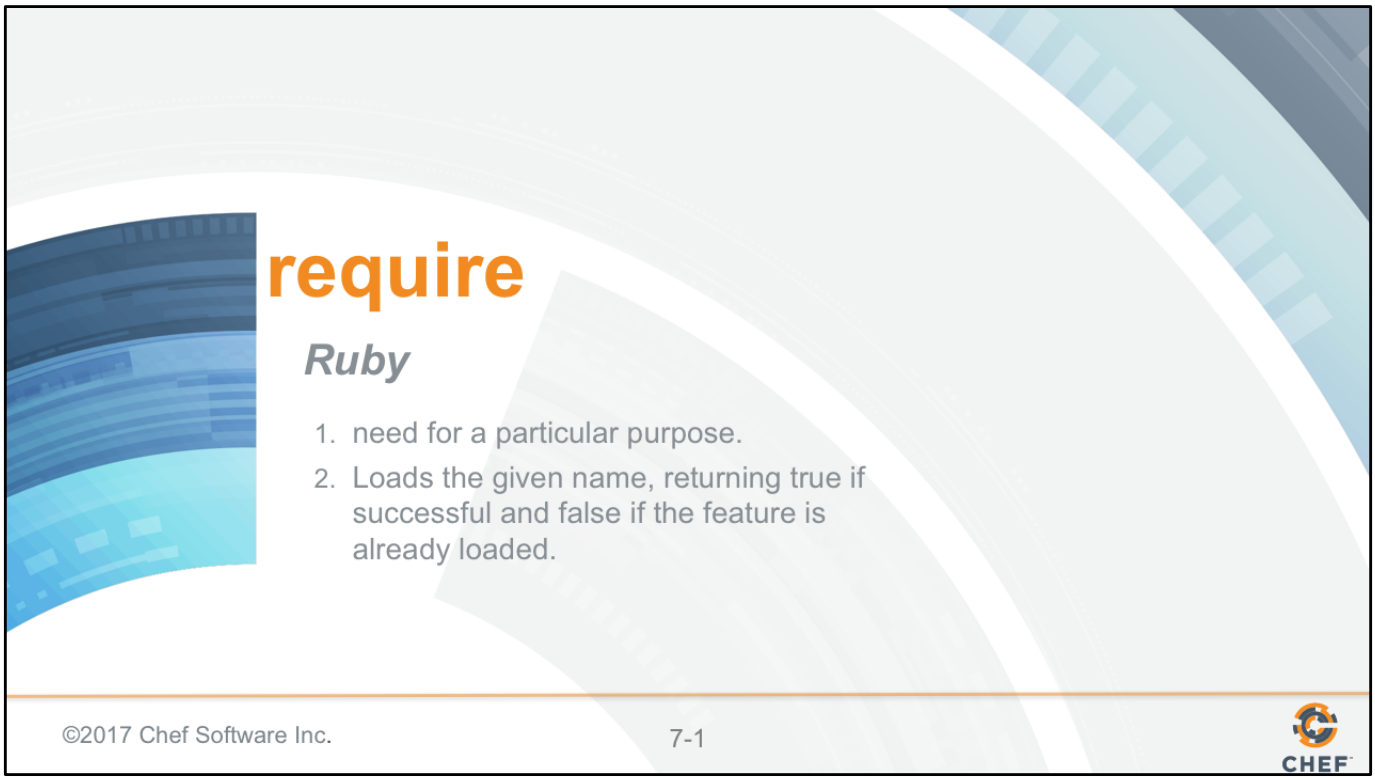


Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

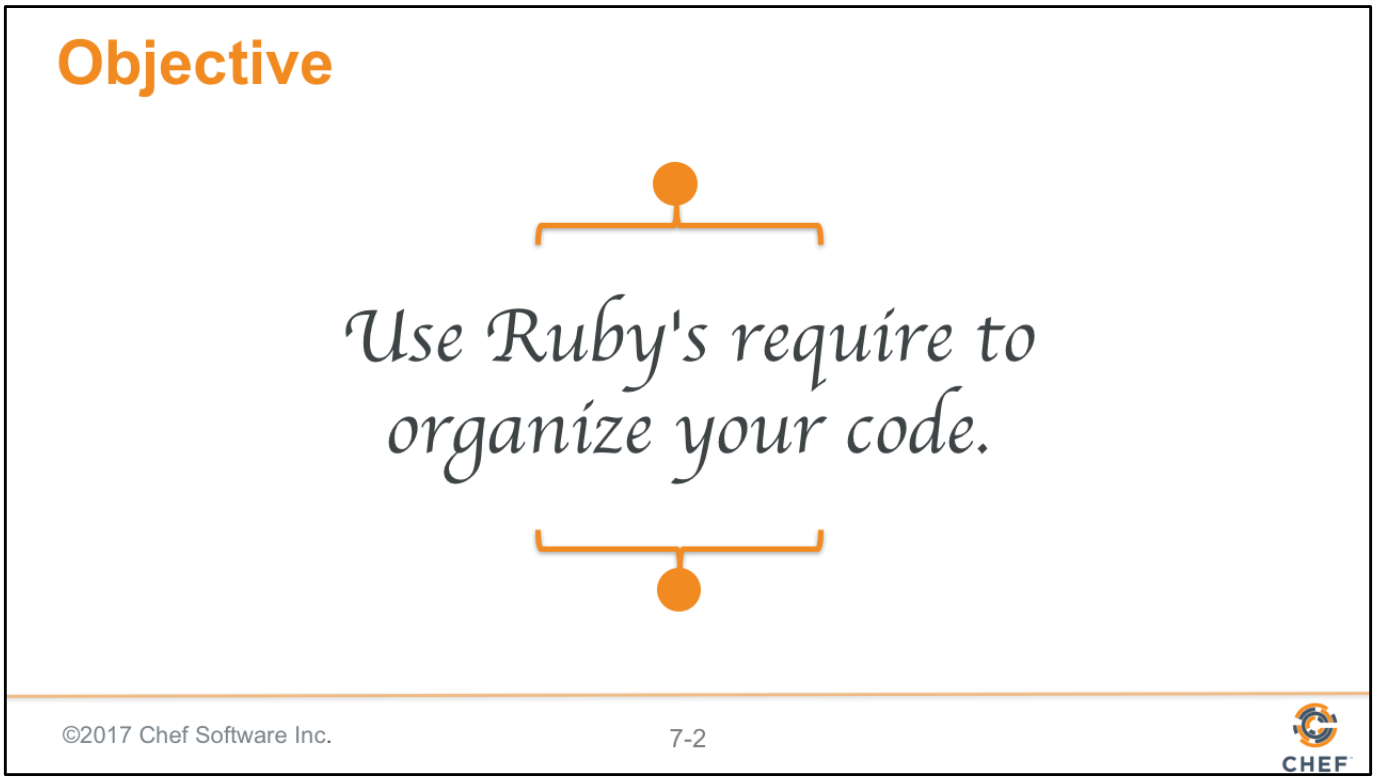
shared_context

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise


CHEF™

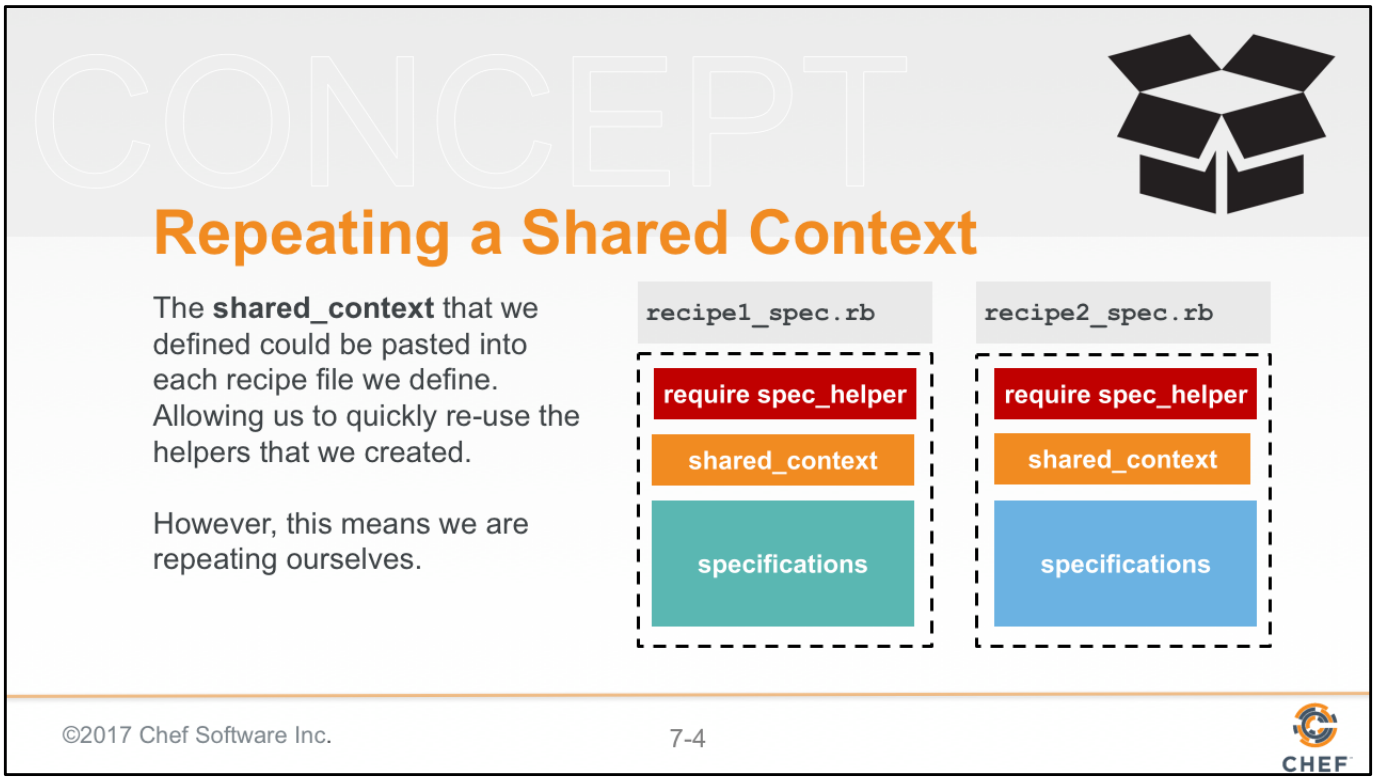


This fifth technique we are going to explore is using Ruby's require method.



The objective of this module is to use Ruby's require method to allow us to organize your code in better ways.

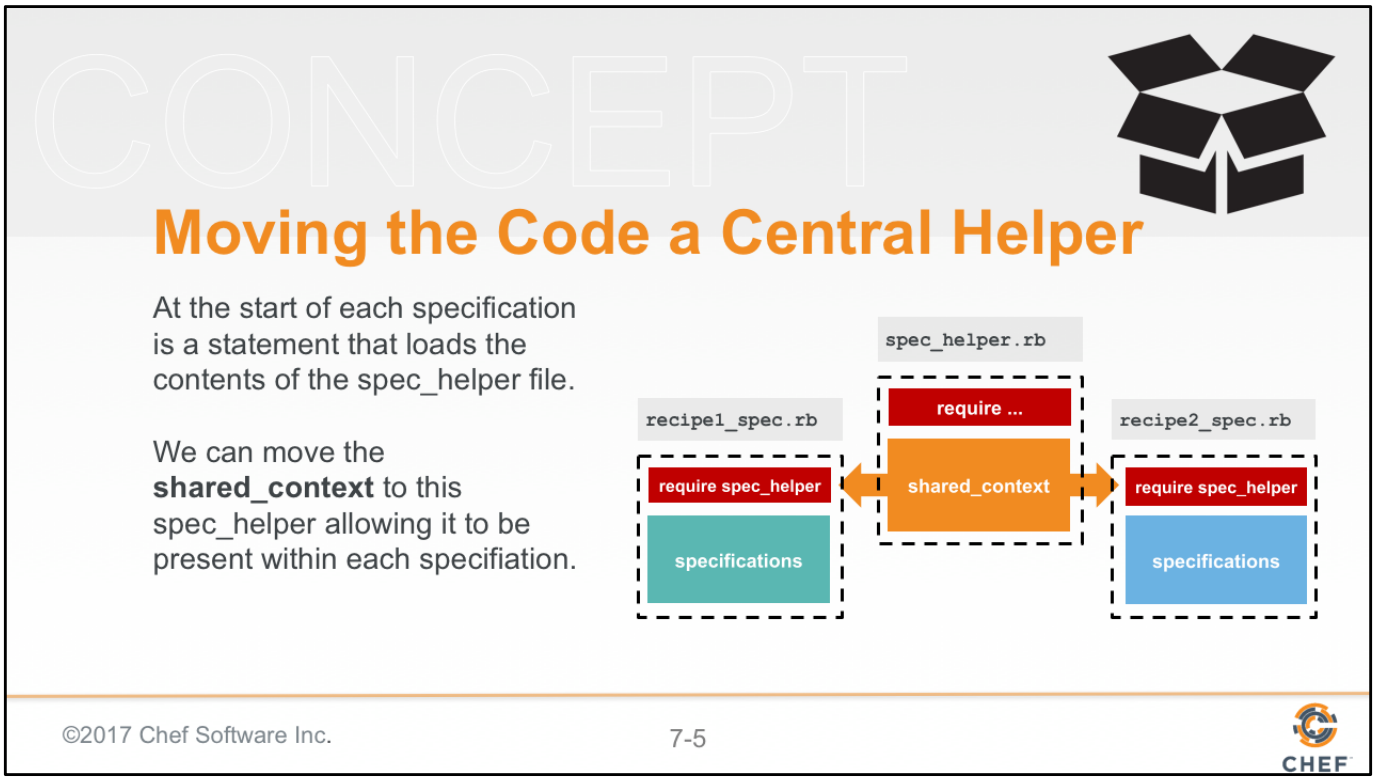
First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.



Having the **shared_context** at the top of the file makes it more difficult to read the specification. All the code to help make the specifications more elegant must exist before it is used below but can be a distraction and decrease a maintainer's comprehension.

But it is incredibly useful when it comes time to define additional specification files. We can copy the shared_context to a new specification, include it, and immediately reap the benefits.

However, copy-and-pasting this **shared_context** still is not a very elegant solution. Imagine if we wanted to fix an issue or add a new helper. We would have to update each one across every file.



At the start of each specification file is a require statement that loads the spec_helper file. This common file itself loads more file. Namely the files that load the ChefSpec libraries and other necessary code.

This file is the perfect location where we can move the **shared_context**. Defining it there will allow it to exist in a single location so we are not repeating ourselves as well as removing all that extra code not related to the expectations that we are expressing.

CONCEPT

require

Loads the given name, returning true if successful and false if the feature is already loaded.

If the filename does not resolve to an absolute path, it will be searched for in the directories listed in $LOAD_PATH ($:).

http://goo.gl/cLKY37

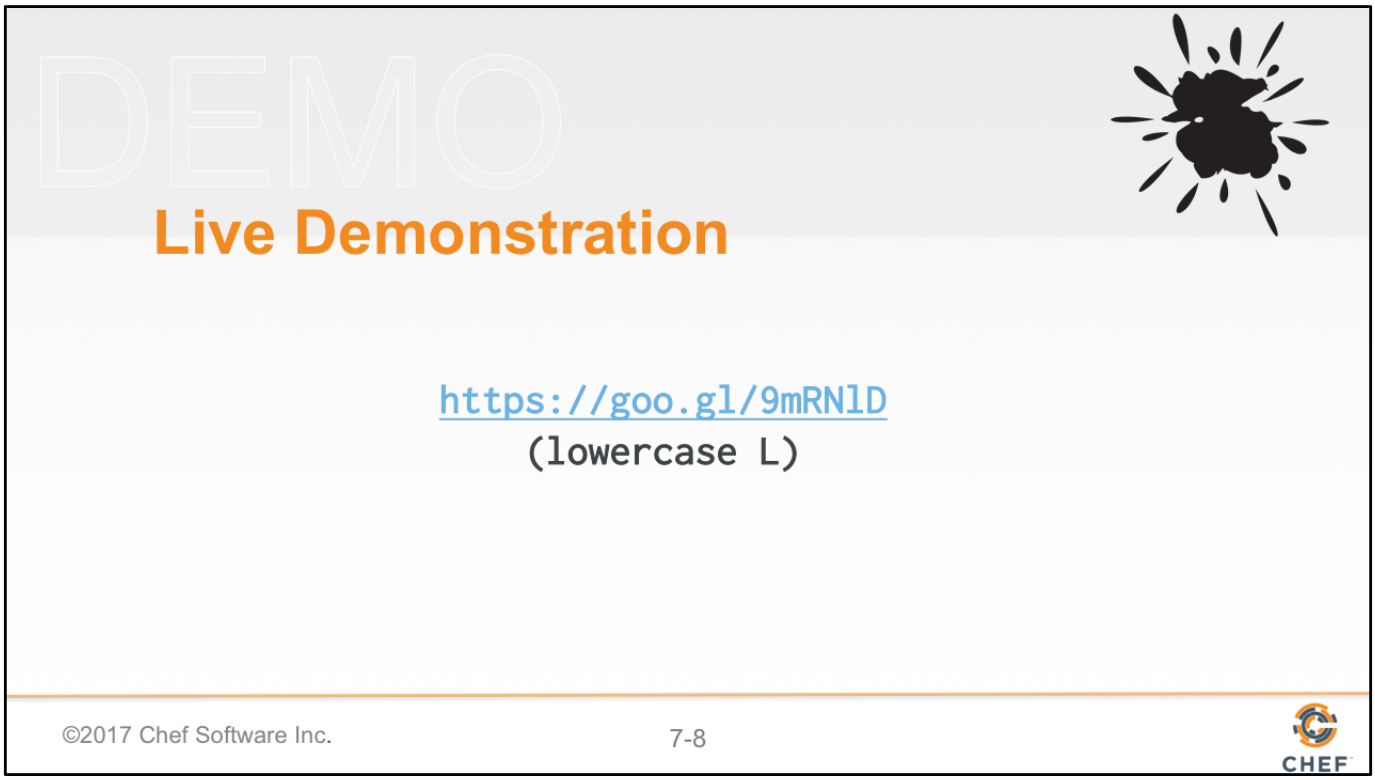
CHEF

require is Ruby method that allows you to load the contents of a specified Ruby file. When you define a require, Ruby appends that file path to each of the file paths found in the $LOAD_PATH global variable.

This $LOAD_PATH is an array of paths and works similar to how $PATH works in most operating systems.

Note: $LOAD_PATH and $: are identical. The first is preferred for clarity. The second is for those that miss writing Perl.

Now it is time to demonstrate how $LOAD_PATH and require work and can be used to help create more elegance for a specification file.



In this demonstration I will show you how to view the contents of the $LOAD_PATH and leverage the existing require of the spec_helper file to increase the clarity and re-usability of the shared_context or any helpers that you haved defined.

require

- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise

Requiring the spec_helper file

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

```
require 'spec_helper'

describe 'ark::default' do
  context 'when no attributes are specified, on an ...platform' do
    let(:chef_run) do
      runner = ChefSpec::SoloRunner.new
      runner.converge(described_recipe)
    end
```


At the top of every pre-generated specification file exists an existing require statement. This require statement is attempting to require the file named spec_helper.rb.

Note: Ruby's require knows that you are loading the contents of a Ruby file so you can omit the extension (.rb).

Note: When Ruby will not attempt to require the same file multiple times. Similar to how Chef's include_recipe will only include the recipe the first time it is included. So when moving dependencies to other files it does not cause an error or circular references.

Viewing the spec_helper file

~/ark/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'

at_exit { ChefSpec::Coverage.report! }

RSpec.configure do |config|
  config.color = true
end

puts $LOAD_PATH
# ... define test helpers and content in this file
```


The spec_helper file is not in the same path as the rest of the recipe specification files. It is located in the root of the spec directory. To understand why this file is able to be loaded correctly we can see the current LOAD_PATH to understand how the file is found and correctly loaded by Ruby.

Uncommenting the following statement will display the current state of the load path variable.

Viewing the $LOAD_PATH

```
> chef exec rspec
```

```
/opt/chefdk/embedded/lib/ruby/gems/2.1.0/gems/uuidtools-2.1.5/lib
/Users/franklinwebber/ark/spec
/Users/franklinwebber/.chefdk/gem/ruby/2.1.0/gems/rspec-support-3.4.1/lib
/Users/franklinwebber/.chefdk/gem/ruby/2.1.0/gems/rspec-core-3.4.4/lib
/opt/chefdk/embedded/lib/ruby/gems/2.1.0/gems/net-ssh-3.1.1/lib
/opt/chefdk/embedded/lib/ruby/gems/2.1.0/gems/fauxhai-3.5.0/lib
/opt/chefdk/embedded/lib/ruby/gems/2.1.0/gems/diff-lcs-1.2.5/lib
/Users/franklinwebber/.chefdk/gem/ruby/2.1.0/gems/rspec-expectations-3.4.0/lib
/Users/franklinwebber/.chefdk/gem/ruby/2.1.0/gems/rspec-mocks-3.4.1/lib
/Users/franklinwebber/.chefdk/gem/ruby/2.1.0/gems/rspec-3.4.0/lib
```


CHEF

When you execute the test suite you will see the current state of the $LOAD_PATH. The $LOAD_PATH is an array of paths. When you use require with a partial path, Ruby will attempt to concatenate that partial path with the paths within this array until it finds a file at the given path.

Note: RSpec automatically adds the spec directory in the current directory to the load path. This is why you need to execute the tests of the root of your cookbook directory, outside of the spec directory. Executing the tests in other locations will often fail as Ruby will often not be able to find the correct path to find the spec_helper.

Adding the shared_context to the spec_helper

~/ark/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'

at_exit { ChefSpec::Coverage.report! }

RSpec.configure do |config|
  config.color = true
end

shared_context 'converging recipe', type: :recipe do
  let(:chef_run) do
    runner = ChefSpec::SoloRunner.new(node_attributes)
    runner.converge(described_recipe)
  end
  # ... the entire shared_context
end
```


We added the shared_context to the spec_helper file.

This **shared_context** now exists in a single location accessible to all specification files.

Removing the shared_context from the spec

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

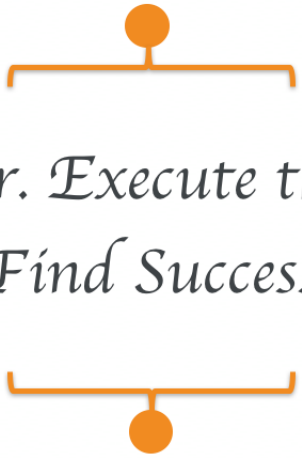
```
require 'spec_helper'

shared_context 'converged recipe' do
 let(:chef_run) do
   runner = ChefSpec::SoloRunner.new(node_attributes)
   runner.converge(described_recipe)
 end

  let(:node_attributes) do
    {}
  end

  def attribute(name) ...
end

describe 'ark::default' do
  context 'when no attributes are specified, on an uns...platform' do
    include_context 'converged recipe'
```


We can now remove the shared_context that is defined at the top of the specification because it now exists in the spec_helper file.

require

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise

Exercise



Refactor. Execute the Tests.
Find Success.

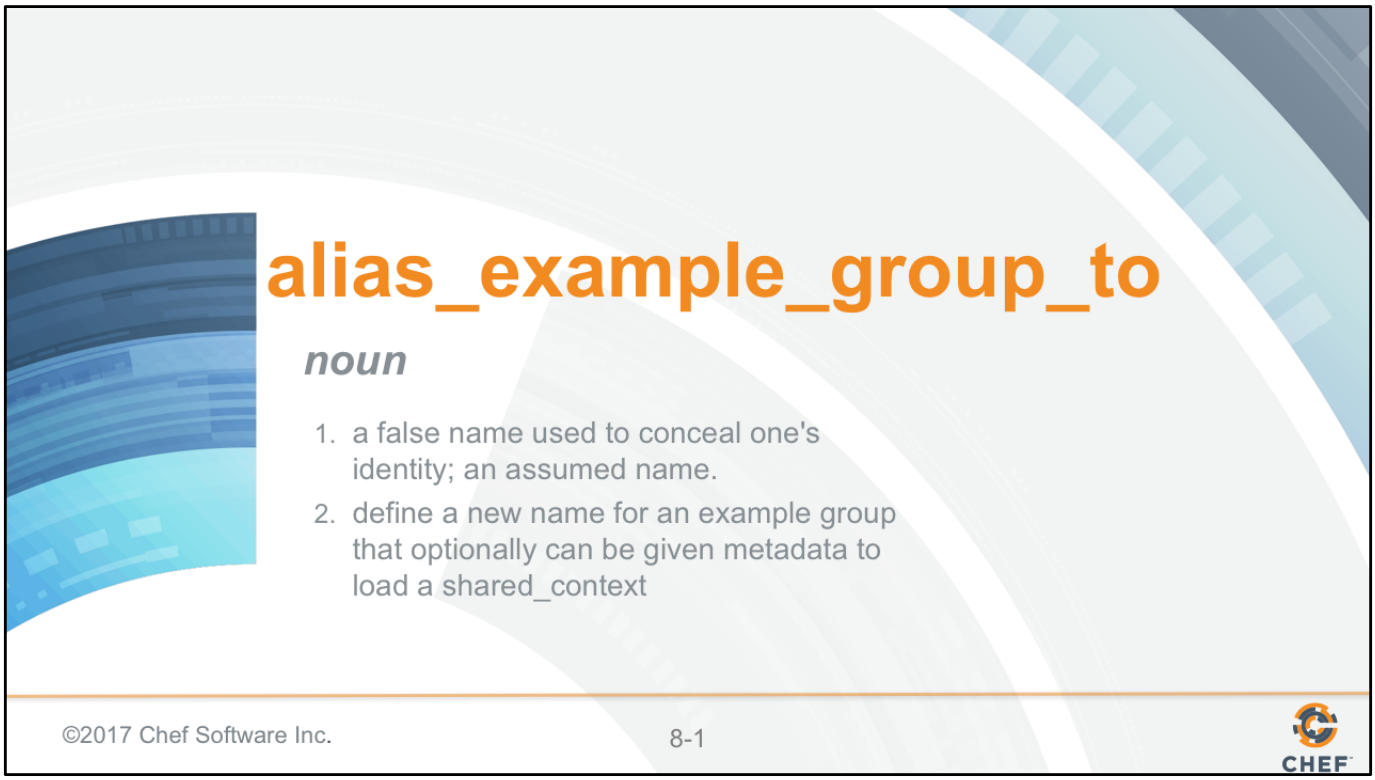
Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

require

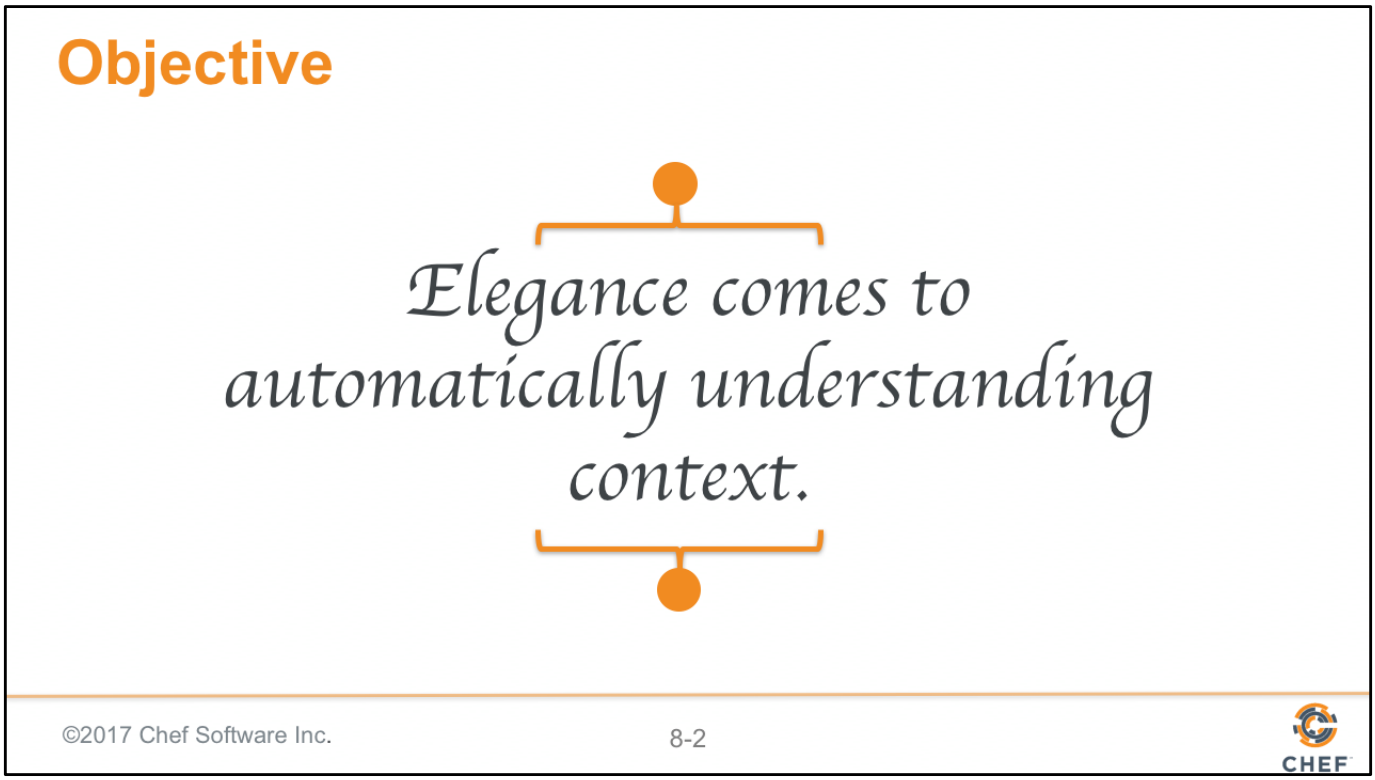
- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise

CHEF


CHEF™



This sixth technique we are going to explore is using alias_example_group_to.



The objective of this module is to use alias_example_group_to allow us to define our own names for an example group and associate that example group with a contextual information.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

Before we began learning about these various techniques we reviewed the concepts behind how RSpec organizes the examples we write to describe the state of our system. Particularly we learned that describe and context were methods we could use to describe the recipe and the various scenarios that we are testing.

It is important to re-iterate that the word 'context' and 'describe' are nearly interchangeable. Defining an example group with 'context' is a personal choice that the community uses to help articulate the various scenarios that are under test. In our case, 'context' is used to help describe the various platforms.

RSpec's alias_example_group_to allows you define your own terms that may better describe the system under test.

CONCEPT

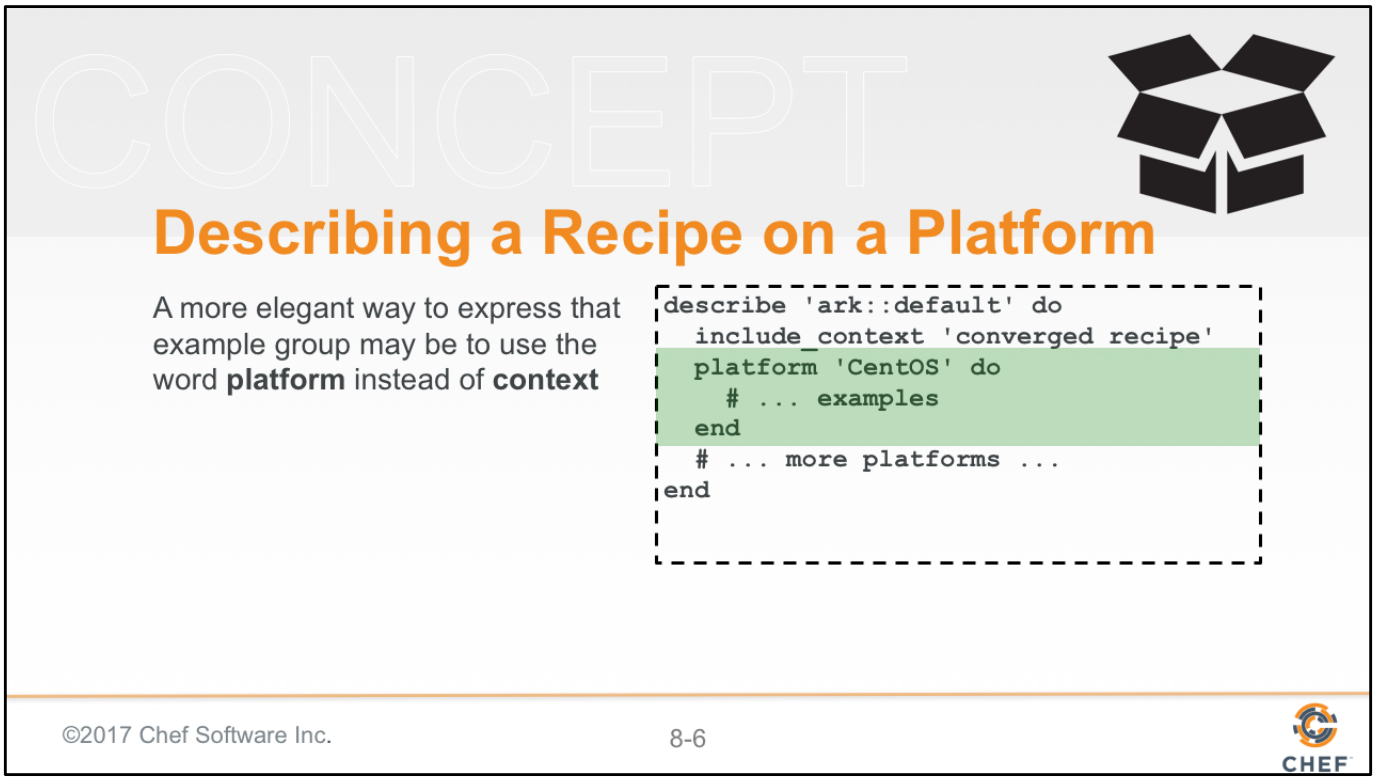
Describing a Recipe on a Platform

Within this recipe's specification every **context** is actually evaluating a platform.

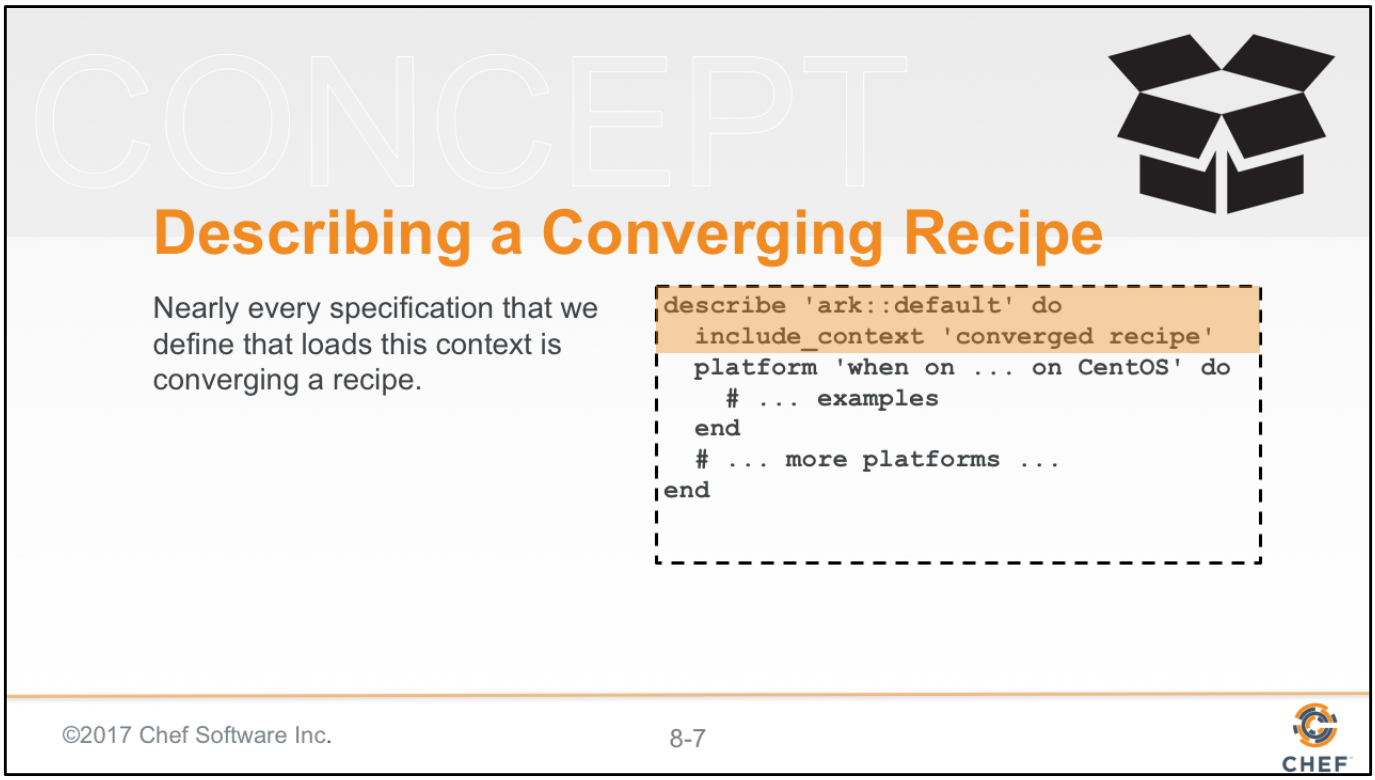
```
describe 'ark::default' do
  include_context 'converged recipe'
  context 'when on ... on CentOS' do
    # ... examples
  end
  # ... more platforms ...
end
```


CHEF

When you examine each **context** within the specification you see that we are assuming default attributes and checking across multiple platforms. The word **context** here really means that each scenario we are interested in is really our platform differences.

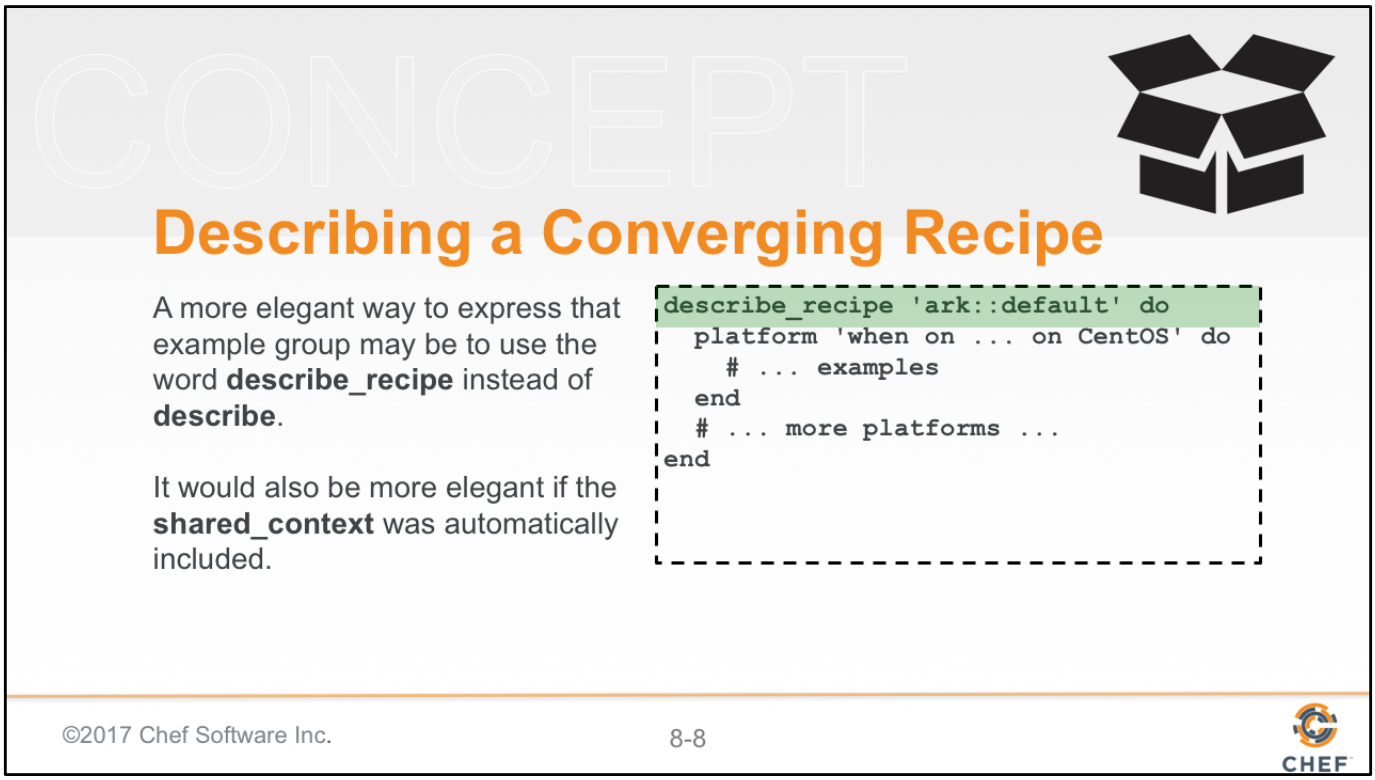


A more ideal word may be **platform** and not **context**. Both convey the same meaning perhaps but given a choice it feels like **platform** does a better job at getting to the point quicker. We can save ourselves more keystrokes and give the future maintainers of this code information more quickly.



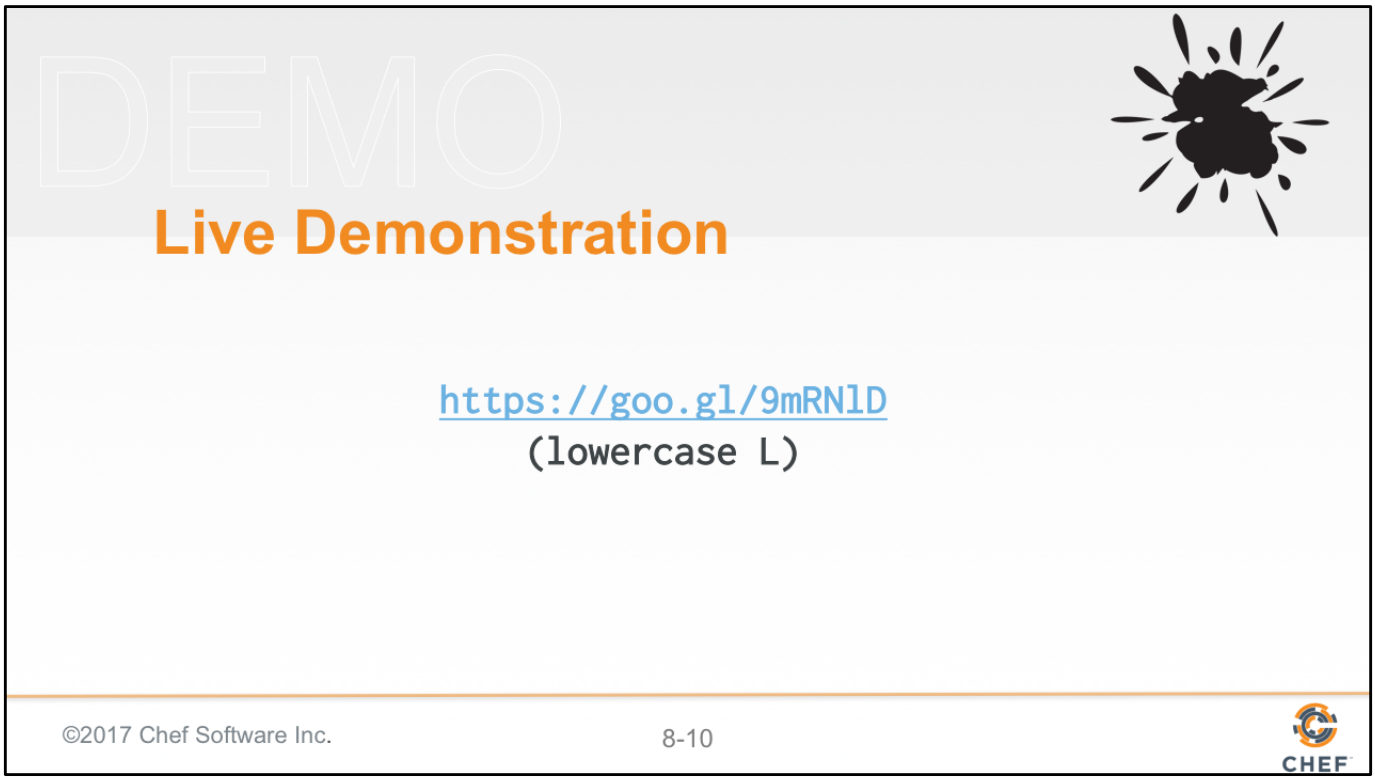
Another situation is the parent example group where we are describing a particular recipe and then saying that this recipe is one that converges. Generally all recipes that are under test are ones that you want to converge.

The first statement is not exactly clear that we are talking about a recipe. The second statement, where we include the context, might be more confusing for future maintainers.



Again it would be useful if we could define an alias that states more clearly that we are defining specifications for a recipe and automatically include the context that we created that contains the helpers that we need.

Now it is time to demonstrate how alias_example_group_to works and how we can use it to further the elegance of our specifications.



In this demonstration I will show you how to define a new alias and then associate metadata with that alias. Then we will refactor our existing example group to use the alias that we have defined.

alias_example_group_to

- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise

Viewing the Current Specification

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

```
require 'spec_helper'

describe 'ark::default' do
  context 'when no attributes are specified, on an ...platform' do
    include_context 'converged recipe'

    # ... examples ...

  end

  # ... other contexts ...
end
```


We started with a specification that used generic example groups that are found in RSpec. RSpec is powerful and allows us to define our own terms for our example groups to help express the clarity.

Viewing the Current Specification

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

```
require 'spec_helper'

describe 'ark::default' do
  include_context 'converged recipe'
  platform 'unspecified platform' do

    # ... examples ...

  end

  # ... other contexts ...
end
```


First we use the example group alias that we wanted to define. Changing every instance of **context** to **platform** as well as shortening the text description.

Defining the platform Example Group

~/ark/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'

at_exit { ChefSpec::Coverage.report! }

RSpec.configure do |config|
  config.color = true
  config.alias_example_group_to :platform
end
```


CHEF

We then added that alias example group to our RSpec configure block within the spec_helper file.

Executing the Tests

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

```
..................

Finished in 3.58 seconds (files took 2.73 seconds to load)
19 examples, 0 failures


ChefSpec Coverage report generated...

  Total Resources:   41
  Touched Resources: 41
  Touch Coverage:    100.0%
```


CHEF

We can execute the tests to ensure that we have declared our alias correctly.

Viewing the Current Specification

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

```
require 'spec_helper'

describe_recipe 'ark::default' do
  include_context 'converged recipe'
  platform 'unspecified platform' do

    # ... examples ...

  end

  # ... other contexts ...
end
```


Again we return to the specification and change the **describe** to **describe_recipe**. We also removed the context that we included.

Defining the alias and tying it to the shared_context

~/ark/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'

at_exit { ChefSpec::Coverage.report! }

RSpec.configure do |config|
  config.color = true
  config.alias_example_group_to :platform
  config.alias_example_group_to :describe_recipe, :type => :recipe
end

shared_context 'converged recipe', :type => :recipe do
```


CHEF

We then returned to the spec_helper file and the RSpec configure block and define a new alias example group. This time we specify some additional metadata that states that this example group is a particular type.

To create the relationship between the **alias_example_group** and the **shared_context** we need to also assign the same type to the **shared_context**.

Executing the Tests

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

```
..................

Finished in 3.58 seconds (files took 2.73 seconds to load)
19 examples, 0 failures


ChefSpec Coverage report generated...

  Total Resources:   41
  Touched Resources: 41
  Touch Coverage:    100.0%
```

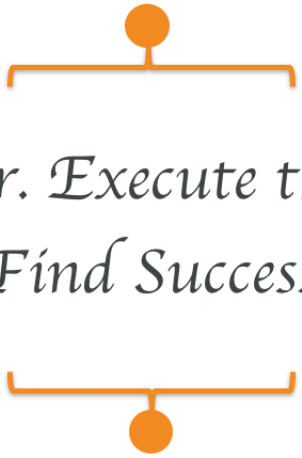

CHEF

We can execute the tests to ensure that we have declared our second alias correctly.

alias_example_group_to

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise

Exercise



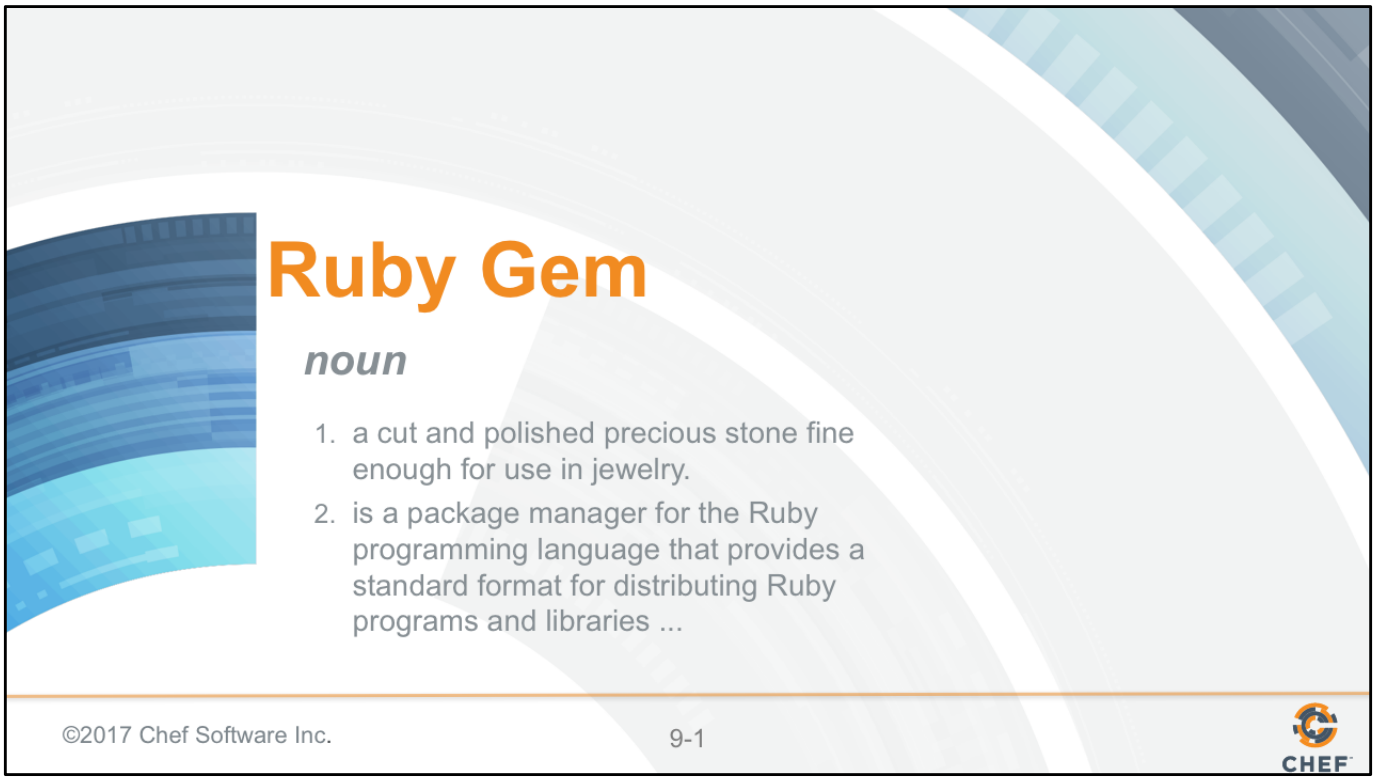
Refactor. Execute the Tests.
Find Success.

Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

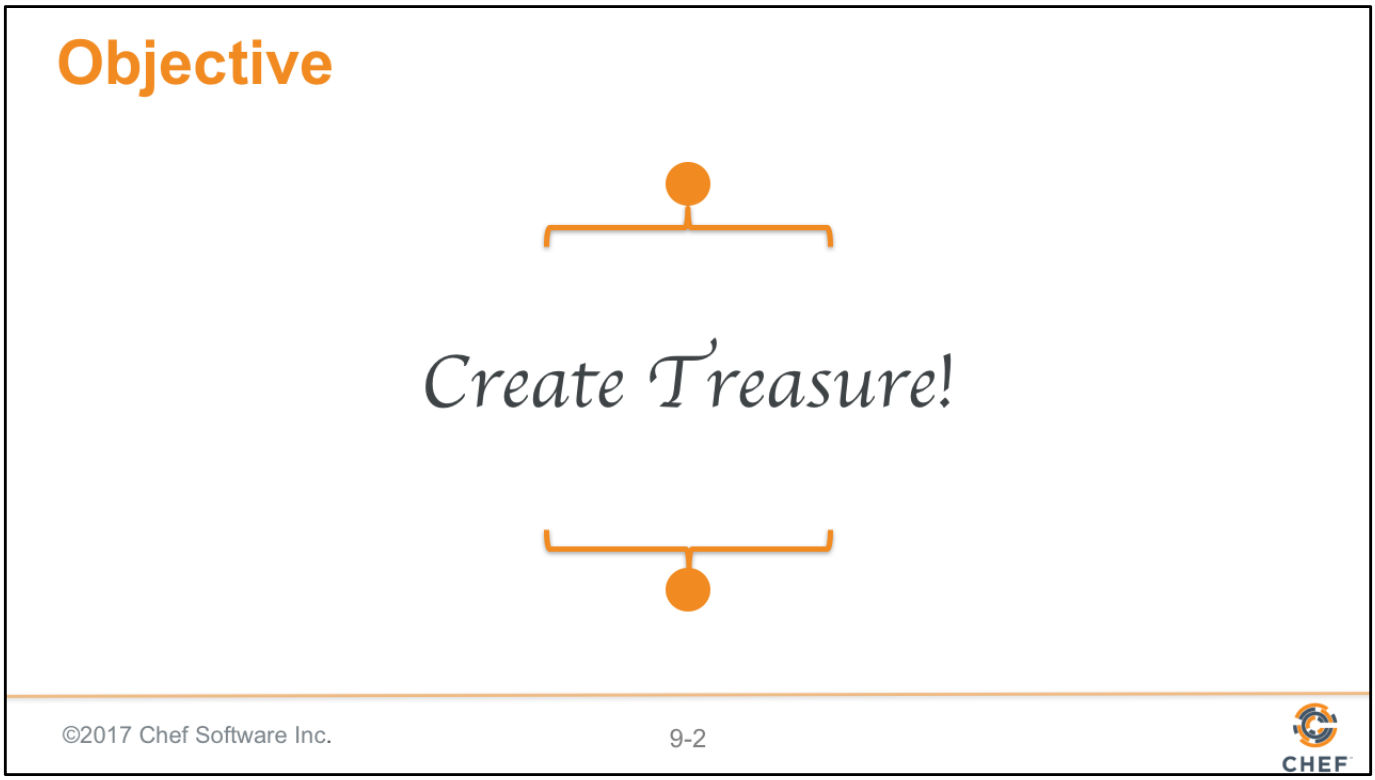
alias_example_group_to

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise


CHEF™



The seventh and final technique we are going to explore is creating a Ruby gem.



The objective of this module is to extract all the helpers we created within this one cookbook into a gem that we can use in another cookbook.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

CONCEPT

RubyGems

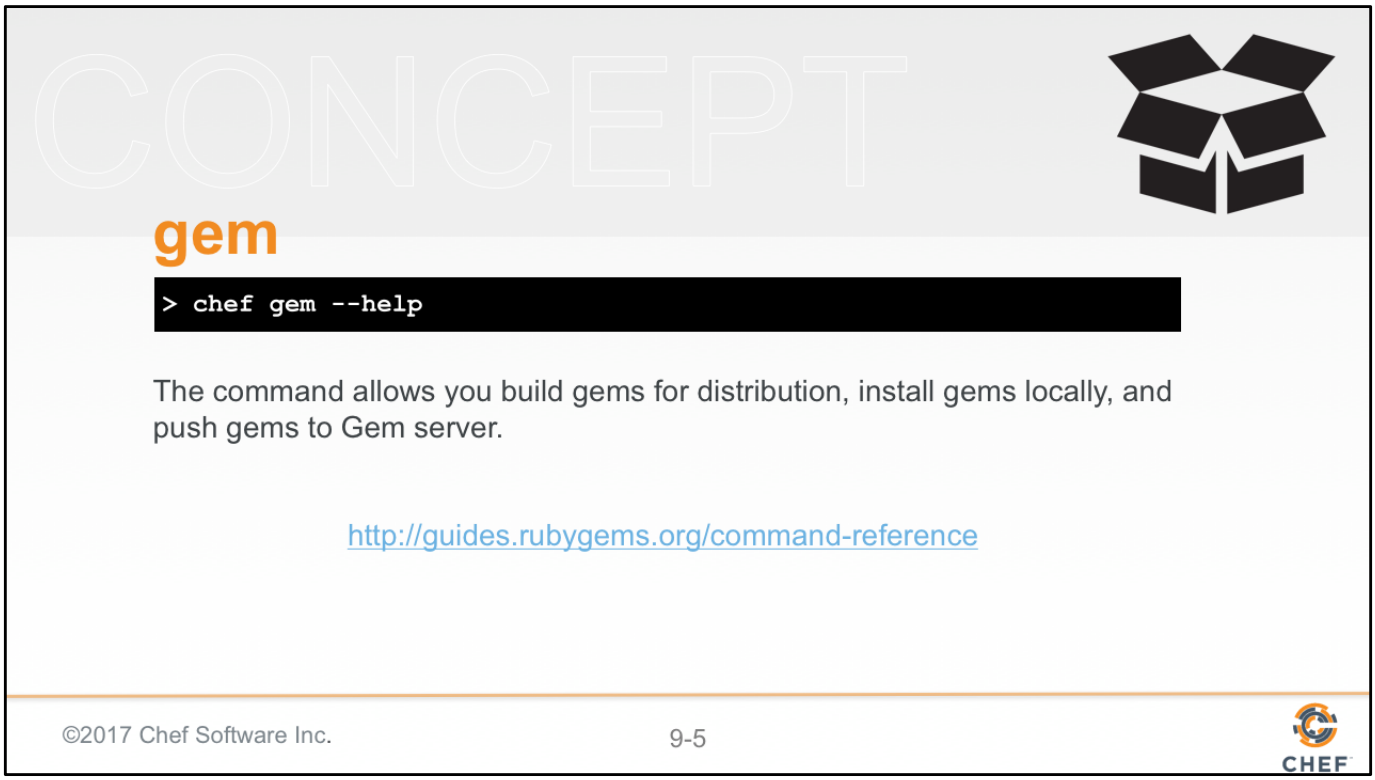
RubyGems is a package manager for the Ruby programming language that provides a standard format for distributing Ruby programs and libraries (in a self-contained format called a "gem"), a tool designed to easily manage the installation of gems, and a server for distributing them.

http://guides.rubygems.org/rubygems-basics

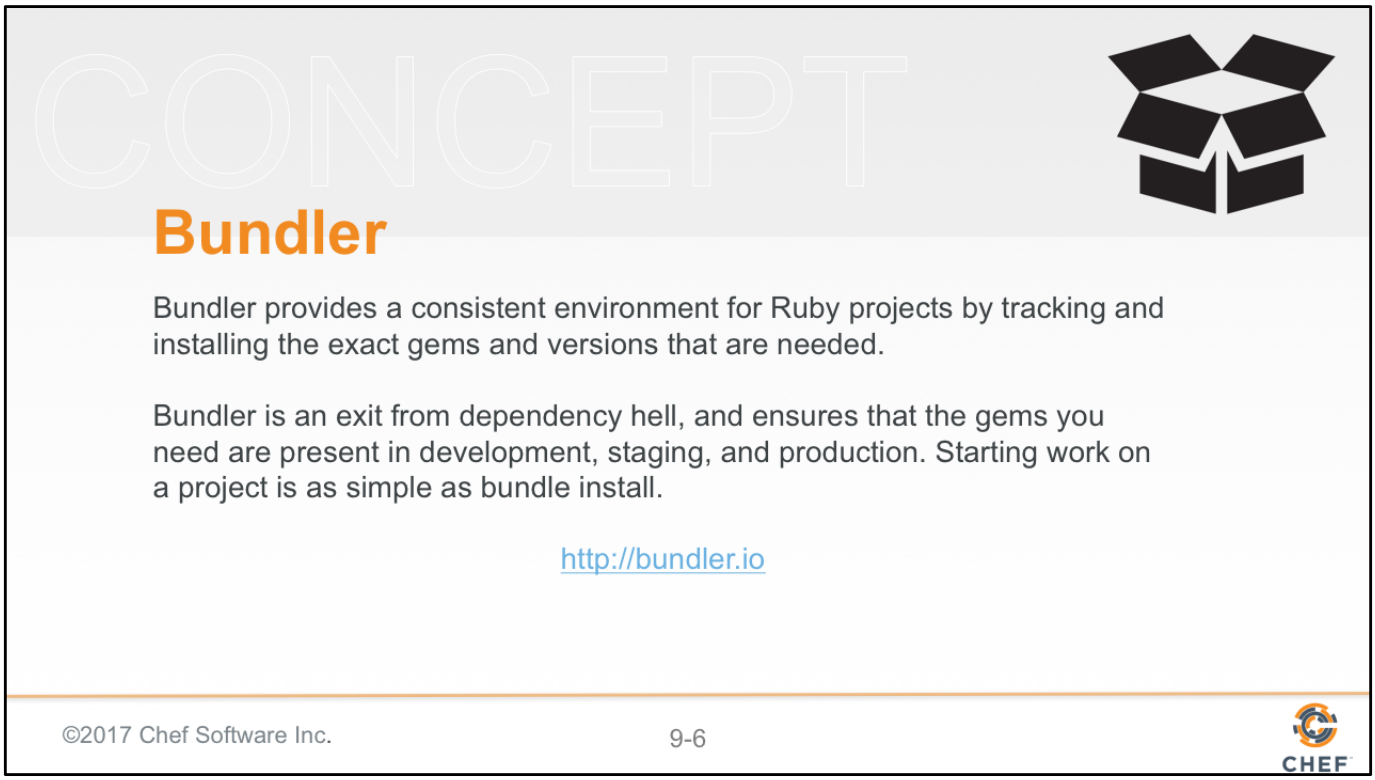
CHEF

A Ruby gem is similar to a Chef cookbook. It is a way for you to package and distribute Ruby code that you would like to share with other projects.

Note: Ruby gems provide classes that you can require into your own Ruby code so it is very close to the concept of a "library cookbook": a cookbook that contains helper methods and custom resources that you employ in your cookbooks.

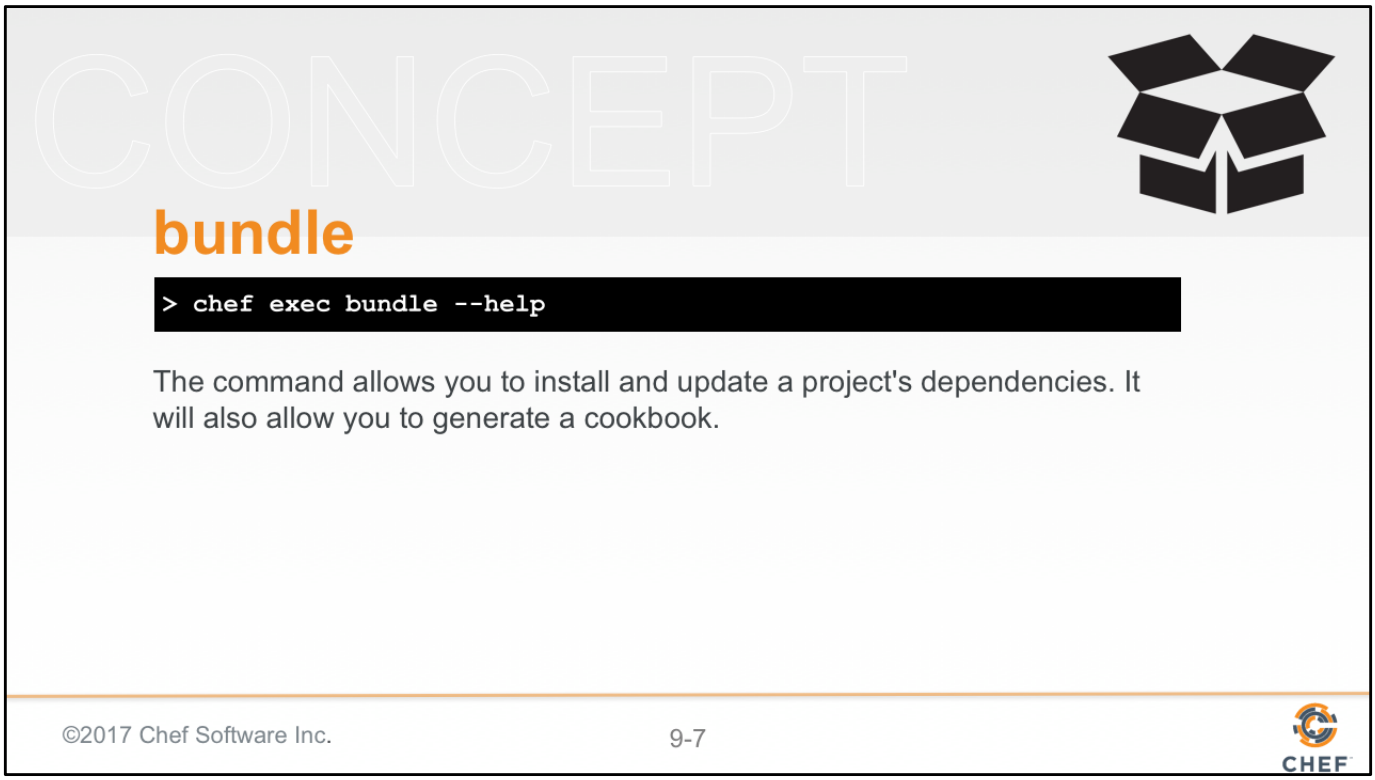


Ruby gems are managed through two tools. The first is the gem tool that allows you to retrieve from Rubygems.org or a custom gem server. You can also install gems locally if you have the archive file already downloaded.



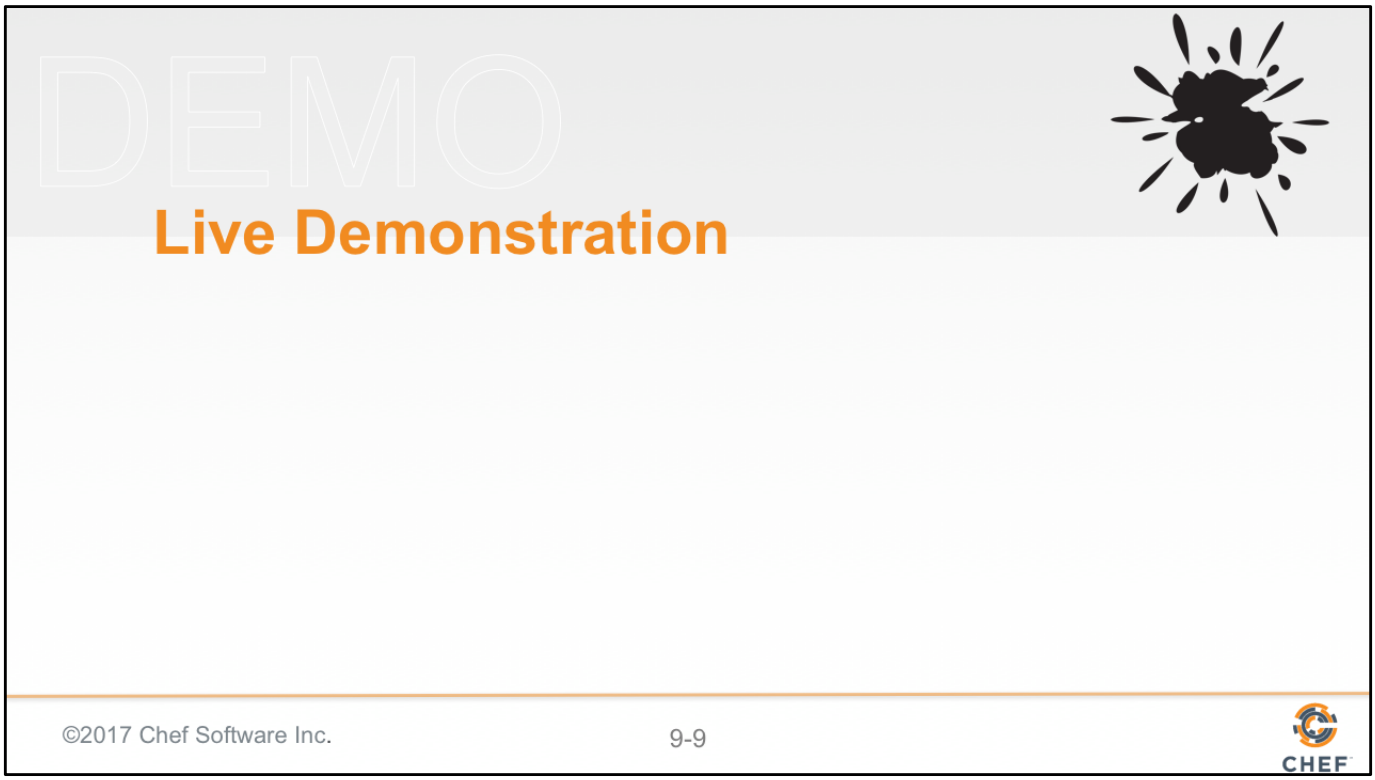
The gem command is useful but when you start working with projects with lots of dependencies it becomes important to use a tool that is capable of resolving dependencies and version constraints. This is where the Bundler tool excels.

Bundler is incredibly similar to Berkshelf from their purpose and even down to some of the commands that you execute.



You can execute the bundler executable through the bundle command. Bundler comes packaged with the Chef Development Kit.

First we will explore the concepts around this technique, I will demonstrate the use of this technique, we will review what was demonstrated, and then I will ask you to participate in a related exercise.

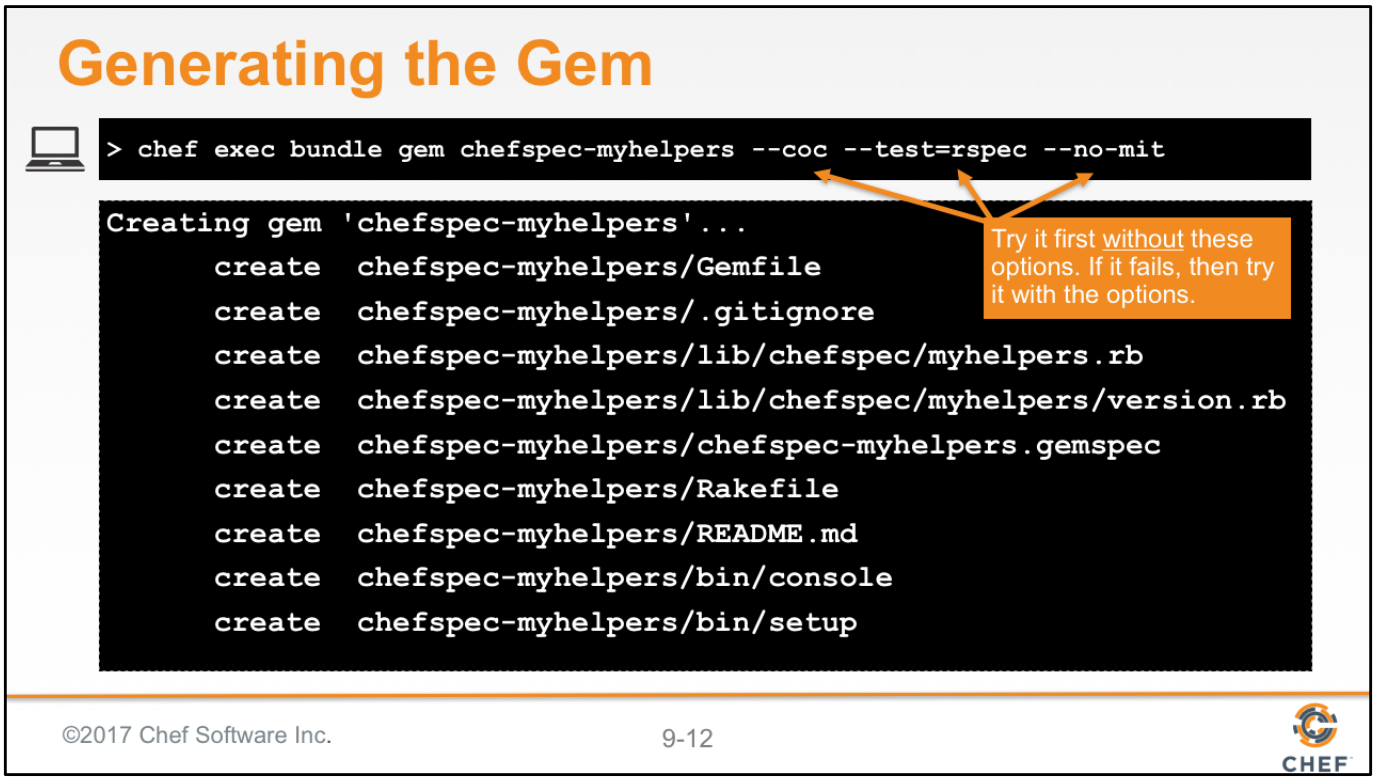


Now it is time to demonstrate how we can create a Ruby gem, migrate the code we have written to the new gem, and then load this new gem into our current cookbook and future cookbooks that want to employ the helpers.

Creating a Ruby Gem

- ✓ Concepts
- ✓ Demonstration
- ❑ Review
- ❑ Exercise

First we returned to the home directory.



Then we used the bundle command to generate a new gem. This command will automatically generate the folder structure with all the files that we need to describe our Ruby gem.

The gem name has a dash between words 'chefspec' and 'myhelpers'. This convention is stating that your gem is an extension of ChefSpec.

Note: This is similar to how we generate cookbooks with the 'chef generate cookbook' command.

Note: The bundle gem command will fail because of an underlying dependency on readline (https://github.com/bundler/bundler/issues/5226) that is not able to handle displaying strings longer than the terminal width. The way around it is to provide all the answers to the questions as flags so that no questions are being asked.

Updating the Gem Specification

~/chefspec-myhelpers/chefspec-myhelpers.gemspec

```
Gem::Specification.new do |spec|
  spec.name          = "chefspec-myhelpers"
  spec.version       = Chefspec::Myhelpers::VERSION
  spec.authors       = ["Franklin Webber"]
  spec.email         = ["franklin.webber@gmail.com"]

  spec.summary       = %q{Provides common ChefSpec Helpers.}
  spec.description   = %q{ChefSpec Helpers that define an alia...}
  spec.homepage      = "https://github.com/burtlo/chefspec...com"
  spec.license       = "MIT"
```


Within the gem that we have created we want to update the gemspec file. The gemspec file is similar to a cookbook's metadata file. In this file you can specify details about the gem.

We edited the summary, description, and homepage with some information to prevent errors when we attempt to package and release this gem.

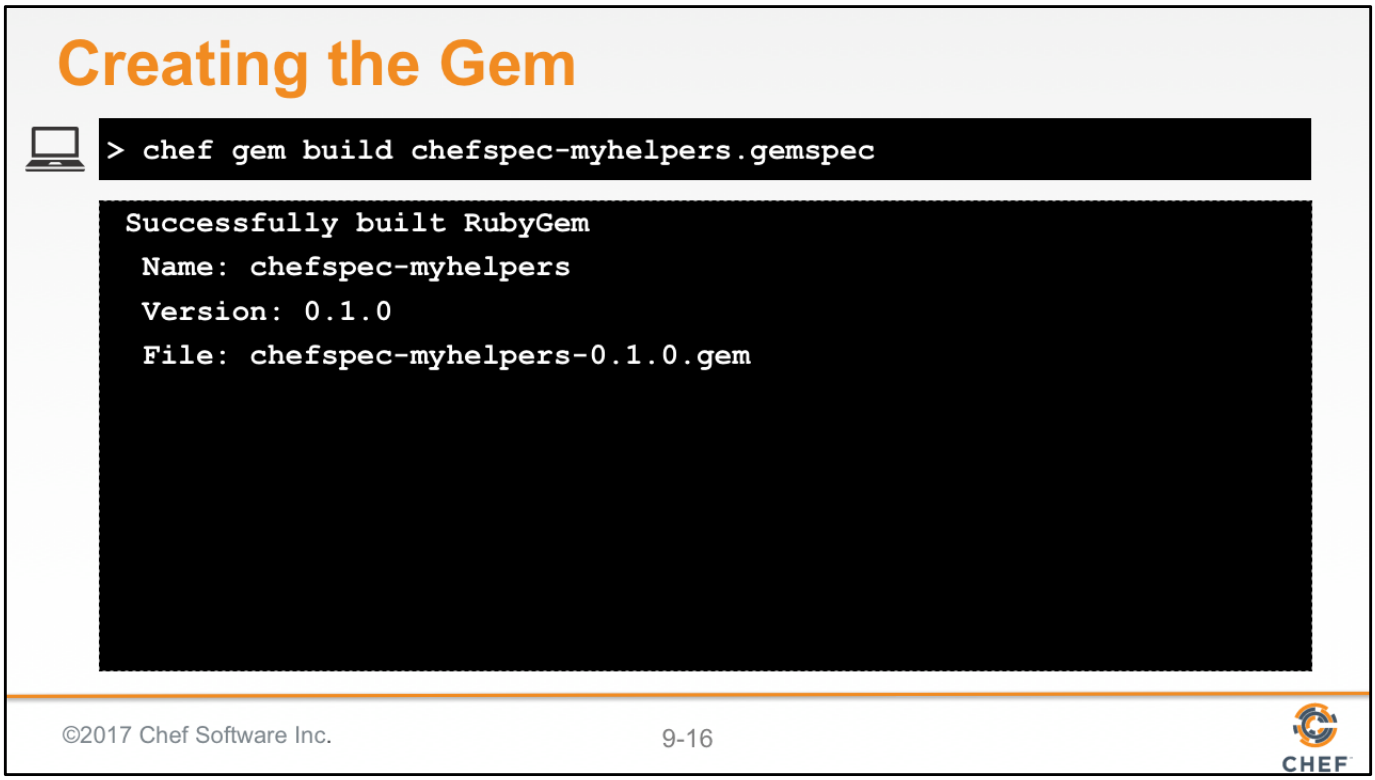
Moving the Shared Context to the Gem

~/chefspec-myhelpers/lib/chefspec/myhelpers.rb

```
RSpec.configure do |config|
  config.color = true
  config.alias_example_group_to :describe_recipe, :type => :recipe
end

shared_context 'converged recipe' do
 let(:chef_run) do
   runner = ChefSpec::SoloRunner.new(node_attributes)
   runner.converge(described_recipe)
 end

  let(:node_attributes) do
    {}
  end
```

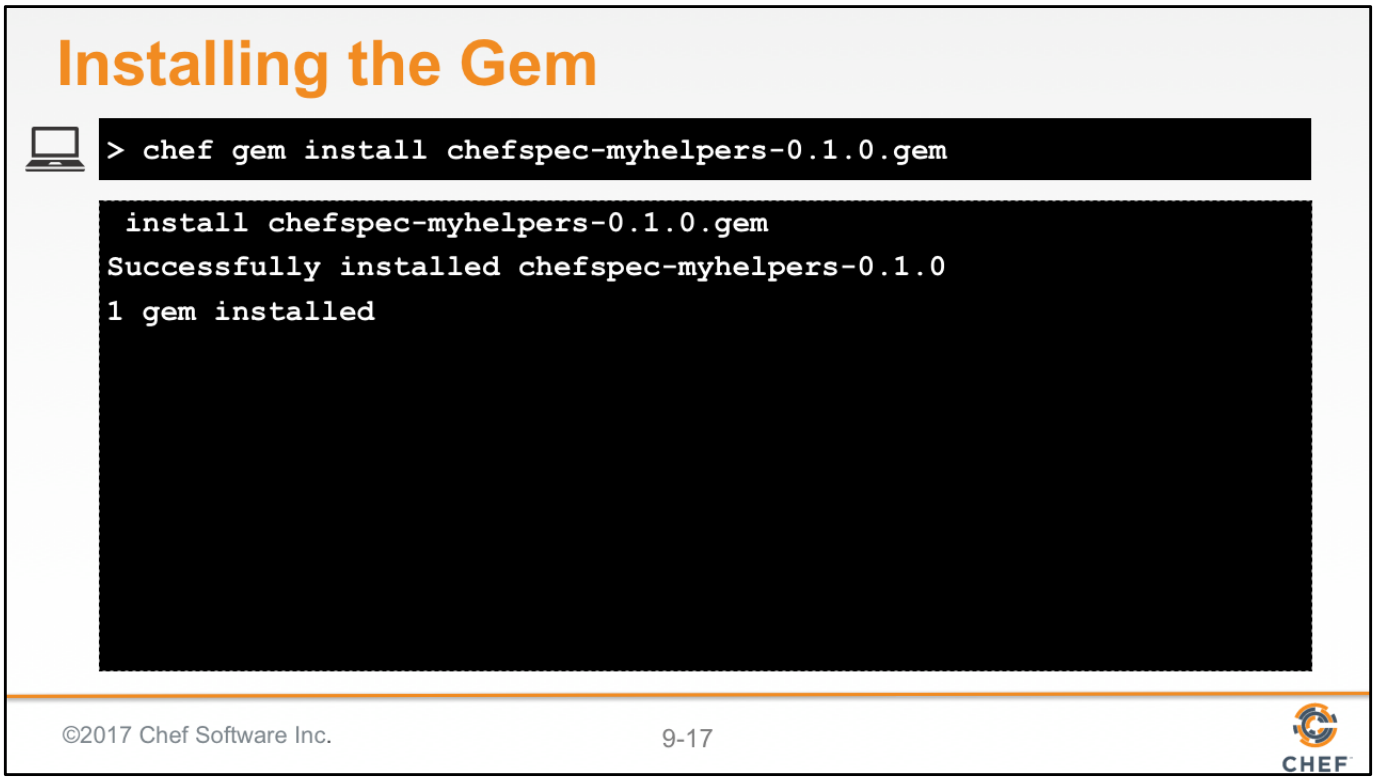

Next we moved all the helpers that we defined in the spec_helper file into the specified file.

We then move into the directory of our new gem.



The gem is then built. This will create an archive version of the gem locally. At this point you can upload the gem to Rubygems.org (along as you setup an account) or you can install this gem locally.

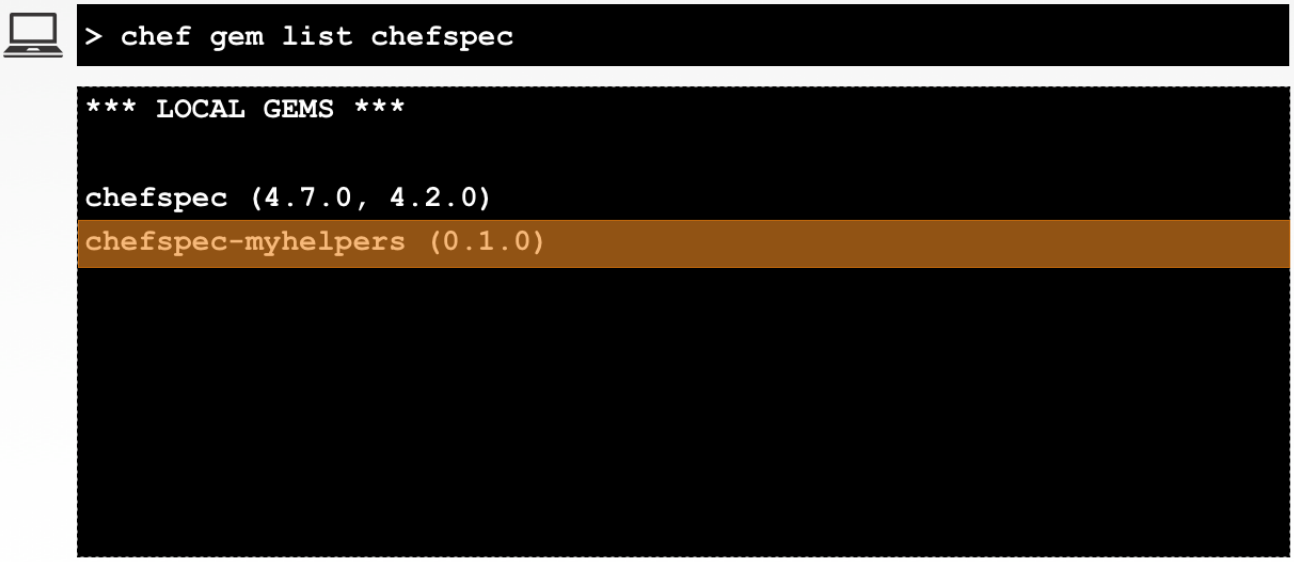
Note: An error may occur with building the gem if some of the fields in the gemspec are not updated to meet certain requirements.



Here we install the gem locally the gem install command. We provide it with the name of the gem archive. The gem is now installed and can be loaded within any of cookbooks.

Note: Distributing gems in this way works on a small scale but ideally you would release this gem to Rubygems.org or your own custom Rubygems server. The other important thing to be aware of is how do you ensure that everyone has this gem. A lot of times that is done through creating a Gemfile within the cookbook to describe the dependencies that are necessary for it to work correctly,

Listing the installed Gem



CHEF

You can verify that the gem exists within the library of gems locally with the gem list command.

Creating a Gemfile for the Cookbook

```
> cd ~/ark && bundle init
```

```
Writing new Gemfile to /home/chef/ark/Gemfile
```


We returned to the cookbook and generated an empty Gemfile. This empty Gemfile is where we are going to specify that our cookbook has a dependency on this new Ruby gem.

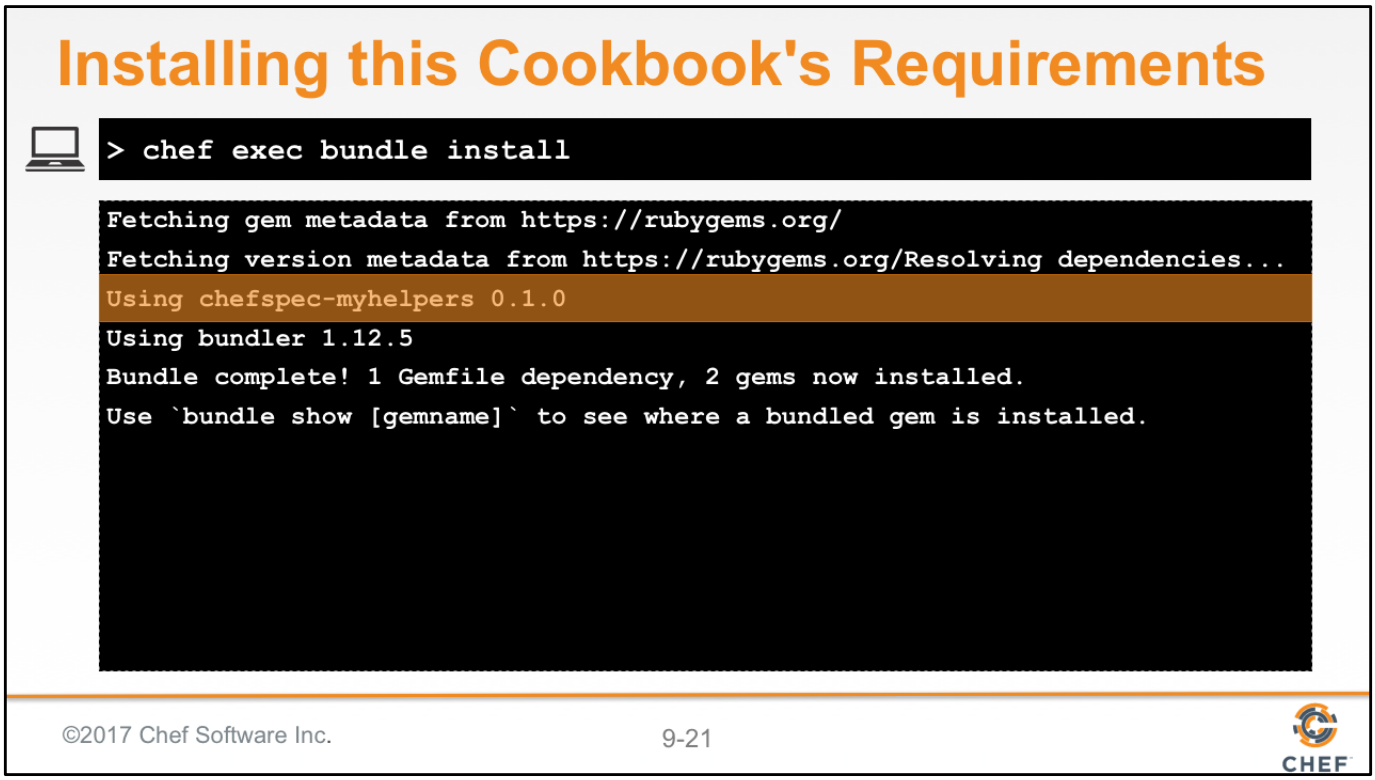
Adding the Helpers Gem to the Requirements

~/ark/Gemfile

```
# frozen_string_literal: true
source "https://rubygems.org"

# gem "rails"                    -
gem 'chefspec-myhelpers'         +
```


We update the Gemfile to include the name of the gem that we are adding. This is similar to when you update a Berksfile to specify a dependency on another cookbook.

Note: The source line is saying that it will look for gems at Rubygems.org. You could specify your own custom gem server as well as a source.



Then you can install the dependencies through this command.

Note: This is not necessary for us as we have already installed the gem locally. However, if you were to release this gem to Rubygems or a custom gem server this is how you could install that dependency.

Updating the Cookbook's spec_helper

~/ark/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'
require 'chefspec/myhelpers'                                    +

at_exit { ChefSpec::Coverage.report! }

RSpec.configure do |config|
  config.color = true
  config.alias_example_group_to :describe_recipe, :type => :recipe-
end

shared_context 'converged recipe', :type => :recipe do          -
```

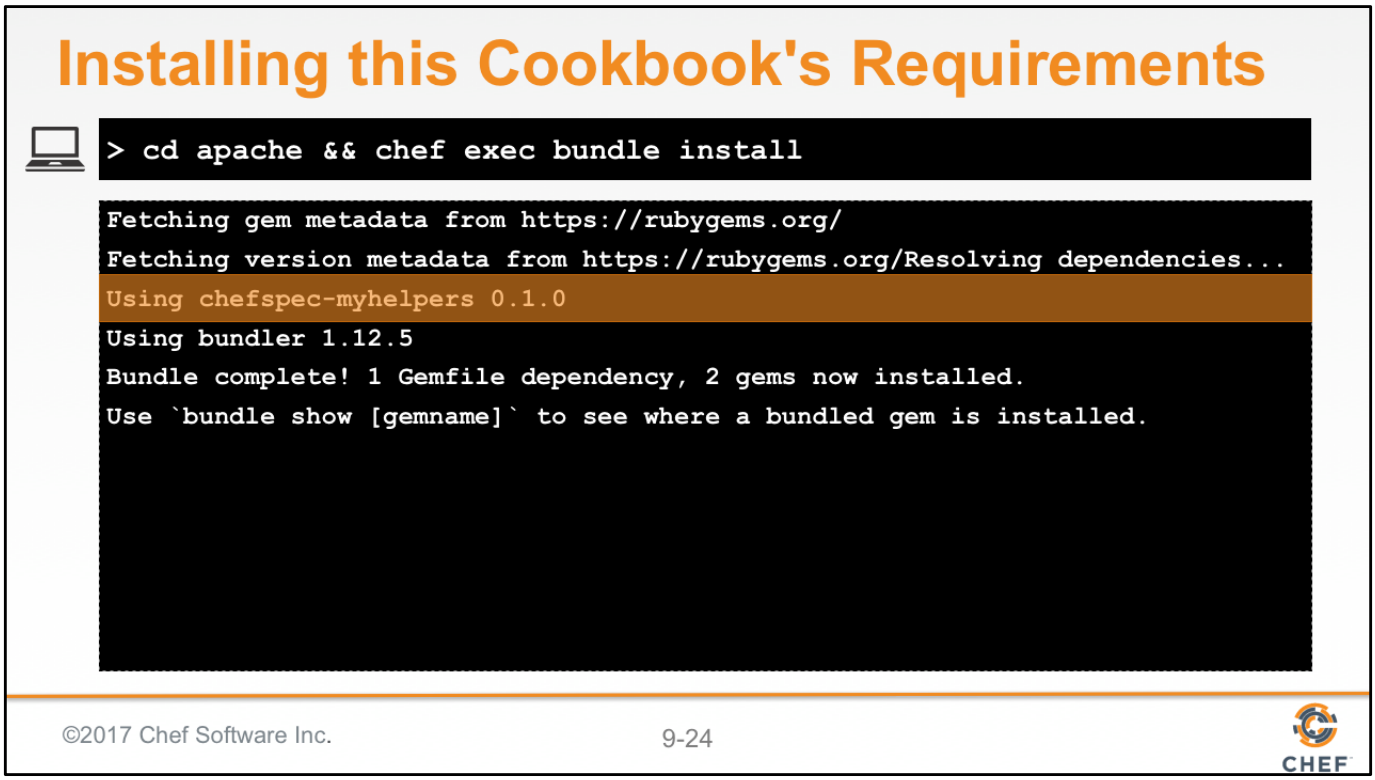

CHEF

Now we want to add a new line which requires the new file that we defined. With all that content in the gem we can remove it from the spec_helper file.

Copying the same Gemfile for a Cookbook

```
cp ~/ark/Gemfile ~/apache/Gemfile
```


There is another cookbook on the workstation that you can use this helper as well. We walked through the same steps that we did for the ark cookbook. We create a Gemfile, populate it, and then install the dependencies.



Again we know we have the dependencies already installed so you really don't have to run this command but it can be useful to get into the habit of doing this when working with cookbooks that contain Gemfiles.

Note: This is not necessary for us as we have already installed the gem locally. However, if you were to release this gem to Rubygems or a custom gem server this is how you could install that dependency.

Adding the Gem to Another Cookbook

~/apache/spec/spec_helper.rb

```
require 'chefspec'
require 'chefspec/berkshelf'
require 'chefspec/myhelpers'

at_exit { ChefSpec::Coverage.report! }
```


CHEF

Because we have this gem installed we can now require it in another cookbook.

Using the Helpers in this New Cookbook

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

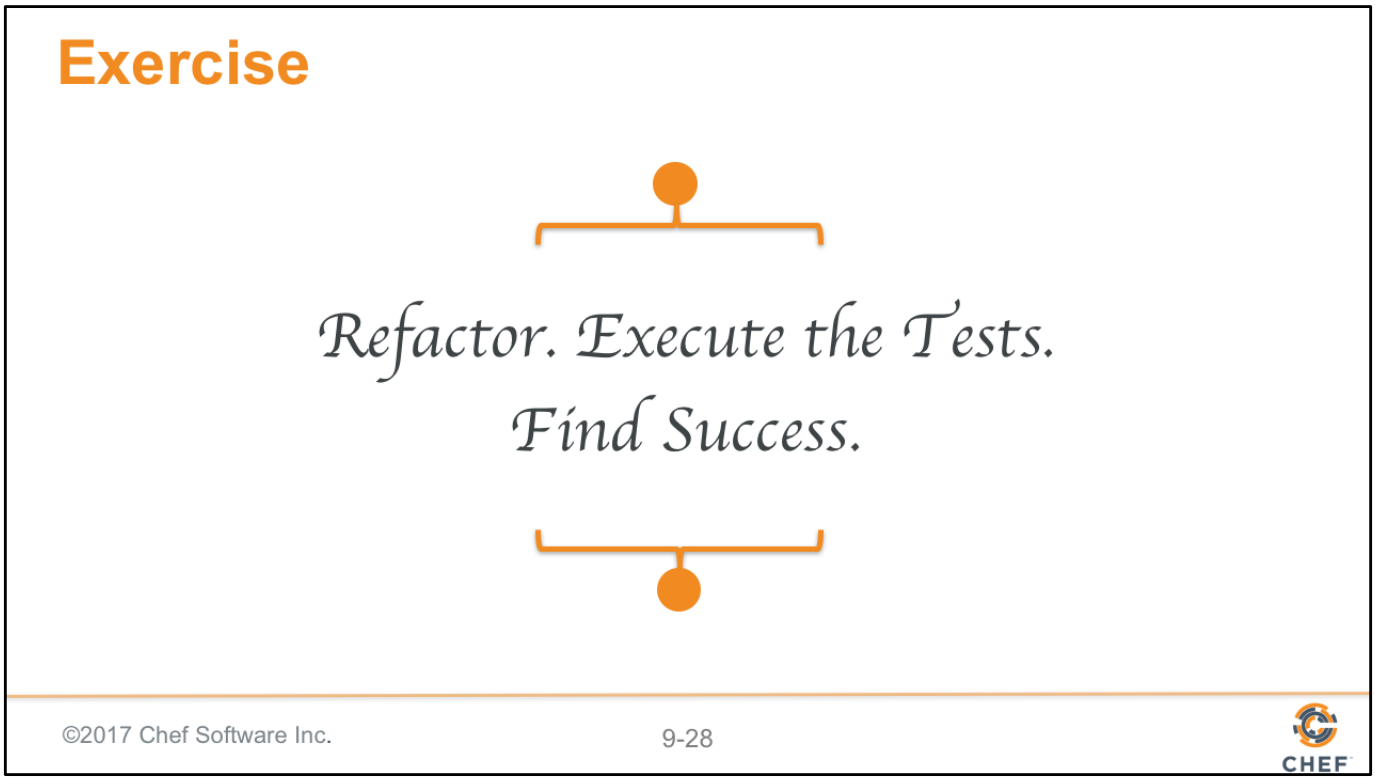
```
require 'spec_helper'

describe_recipe 'apache::default' do
  platform 'unspecified platform' do
    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    let(:installed_packages) { %w[ httpd ] }
    it_behaves_like 'installs packages'

  end
end
```


CHEF

We can immediately see the benefit of our helpers within this new cookbook. We can very quickly write expectations about our installed packages as well as set new details about platforms.

Creating a Ruby Gem

- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ❑ Exercise



Now it is your turn. Refactor the code with this technique. Execute the tests. Find success.

Creating a Ruby Gem

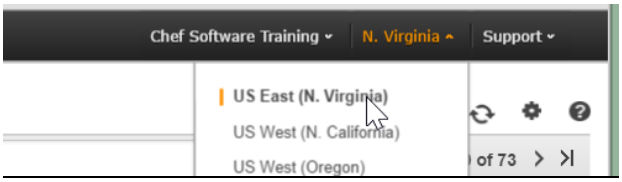
- ✓ Concepts
- ✓ Demonstration
- ✓ Review
- ✓ Exercise


CHEF™

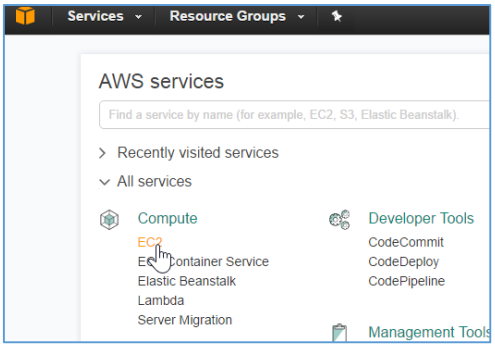
Appendix Z: Course-wide Instructor Notes

1. Training Lab System Setup

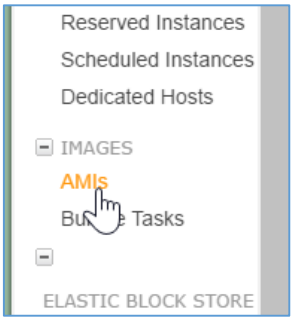
1. Open the AWS site from here: https://aws.amazon.com/
   - Login Credentials for Chef instructors: training-aws@chef.io
   - Password: Contact Chef Training Services if you don't know it or how to obtain it. training@chef.io
   - Partner credentials should be provided by Chef directly to partners.
   - Ensure you are viewing the US East (N. Virginia) region since these AMIs reside there.



2. You may need to expand **All Services** and then click the **EC2** link shown below since the AWS UI may have changed since your last visit.



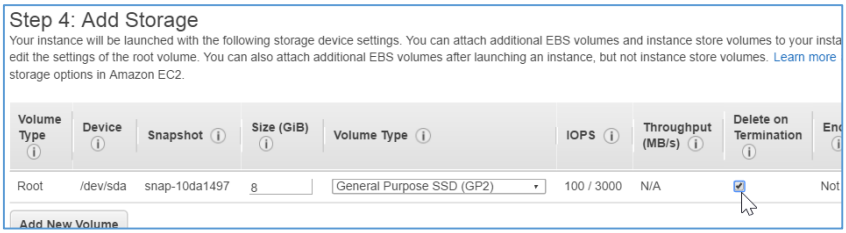
3. From the navigation pane on the left, select **Images > AMIs**. A page will display with a list of available AMIs.



4. Select **Elegant Tests - CentOS 6.7 - 1.2.0** (ami-b519b8a3) from the list of options.
5. Click **Launch**. The "Step 2" page displays.
6. Select the first **Micro Instance** from the list provided and click **Next: Configure Instance Details** at the bottom of the screen. The "Step 3" page displays.
7. Enter the **Number of Instances**.

> *Note: You will need 1 instance for each student enrolled in the class – and 1 for yourself. This instance is a workstation that has Chef installed and all the code for the exercises.*

8. Click **Next: Add Storage** at the bottom of the page. The "Step 4" page displays. Check the **Delete on Termination** check box.



9. Click **Next: Tag Instance** at the bottom of the page. The "Step 5" page displays.
10. Enter a **Value**.

    Note: A recommended naming convention for the instances: [TRAINER'S INITIALS] - [CLASS NAME] - [CLASS DATE]

11. Click **Next: Configure Security Group**. The "Step 6" page displays.
12. Click the **Select an existing security group** radio button. A list of security groups displays.
13. Select **all-open**.
14. Click **Review and Launch** at the bottom of the screen. The "Step 7" page displays.
15. After you review the instances, click **Launch**. The "Select a key pair" window displays.
16. Select **Proceed without a key pair** from the drop down menu and click the acknowledgement check box. (**Note**: Optionally, you could select your own key pair if you want to ensure you can log into an instance in case a student changes the predefined password.)
17. Click **Launch Instances**. The "Launch Status" page displays.
18. Click **View Instances**. The instances list displays.
19. From here, copy all of the instances and create a gist file to share with the class.
20. Use goo.gl to shorten the URL to the gist file.

You could also use this spreadsheet to track and distribute IP addresses if you know how to copy and share such a spreadhseet : http://bit.ly/1MYHVc3

Note: The login credentials and password for the AMIs used in class are chef/chef. You'll need to tell the students that at the appropriate time.

Please remember to terminate your instances when the class is completely done.