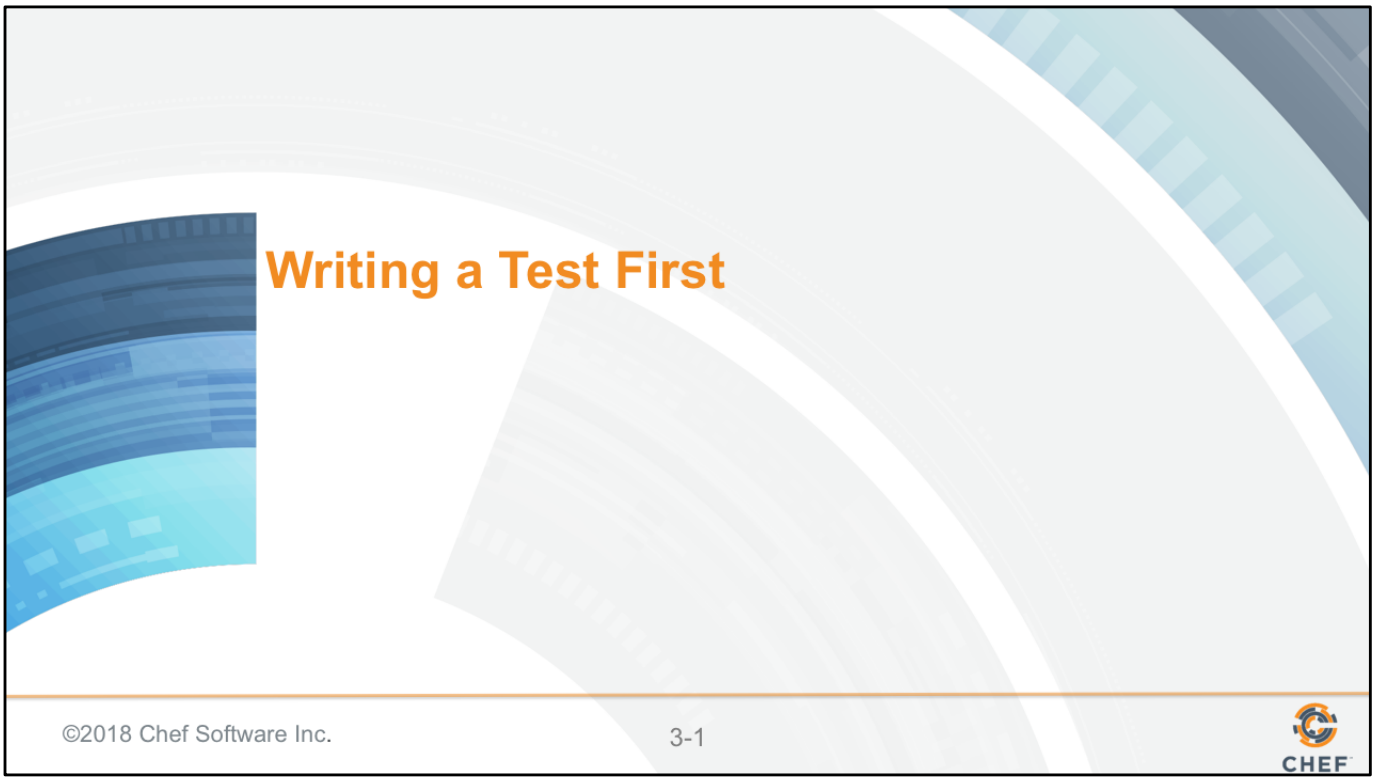
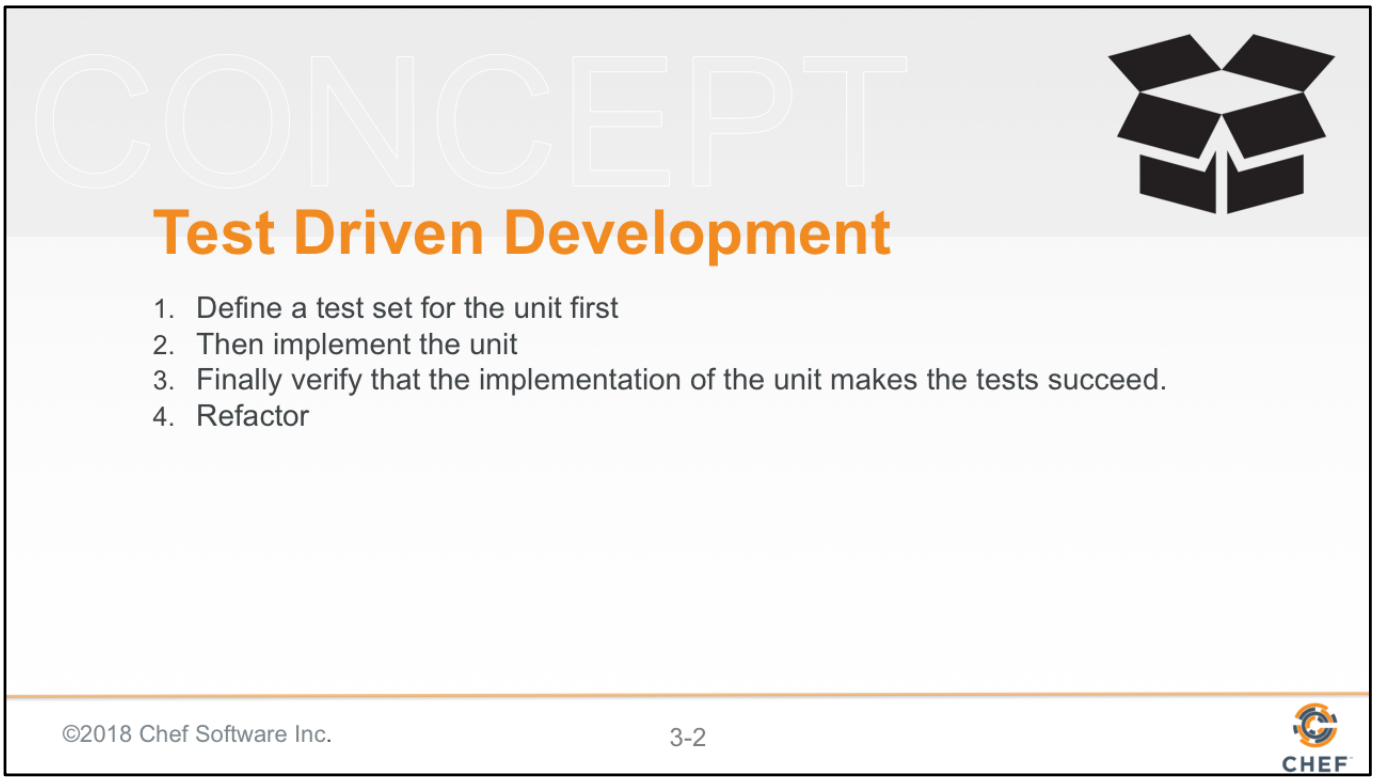


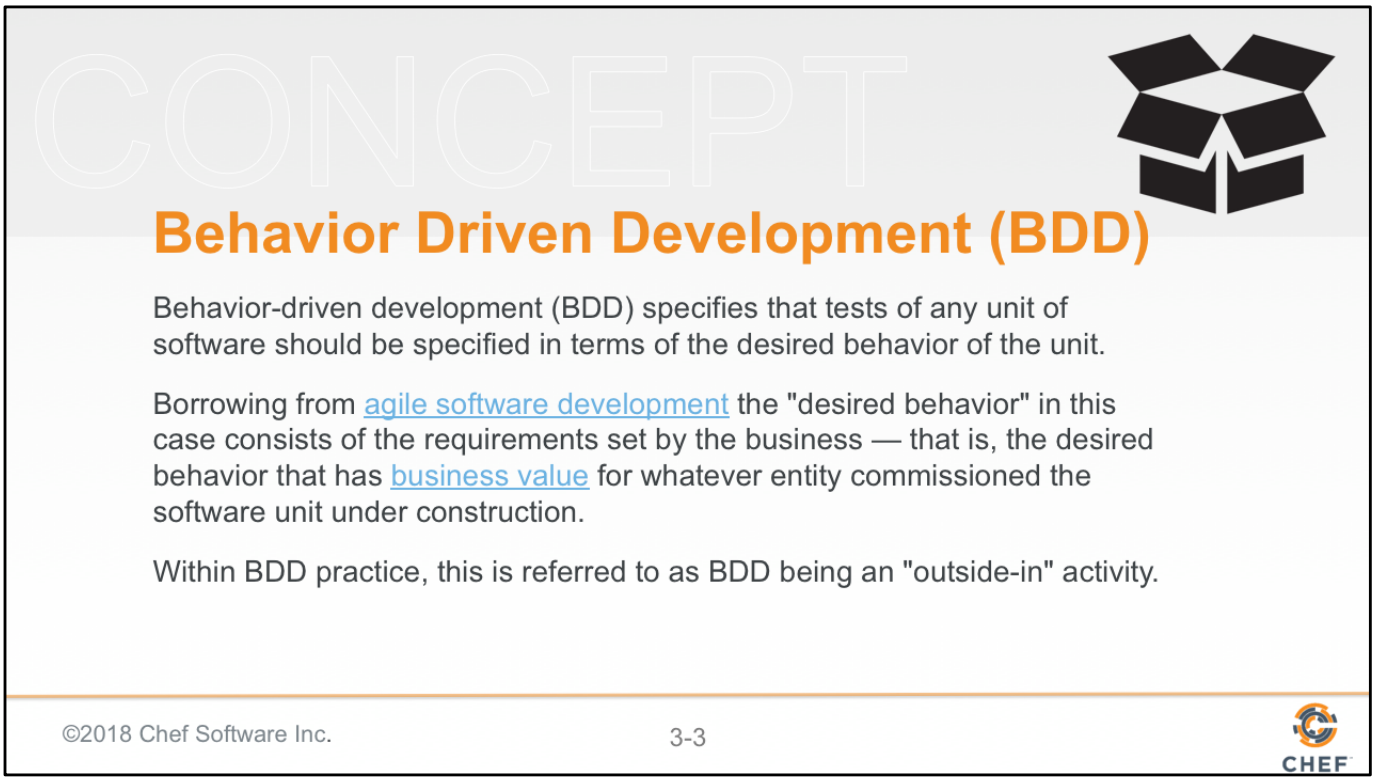
Writing tests are often difficult. Writing tests before you have written the code that you want to test can often feel like a leap of faith. An act that requires a level of clairvoyance reserved for magicians or con-artists. Some have likened it towards starting a story by first writing the conclusion.



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.

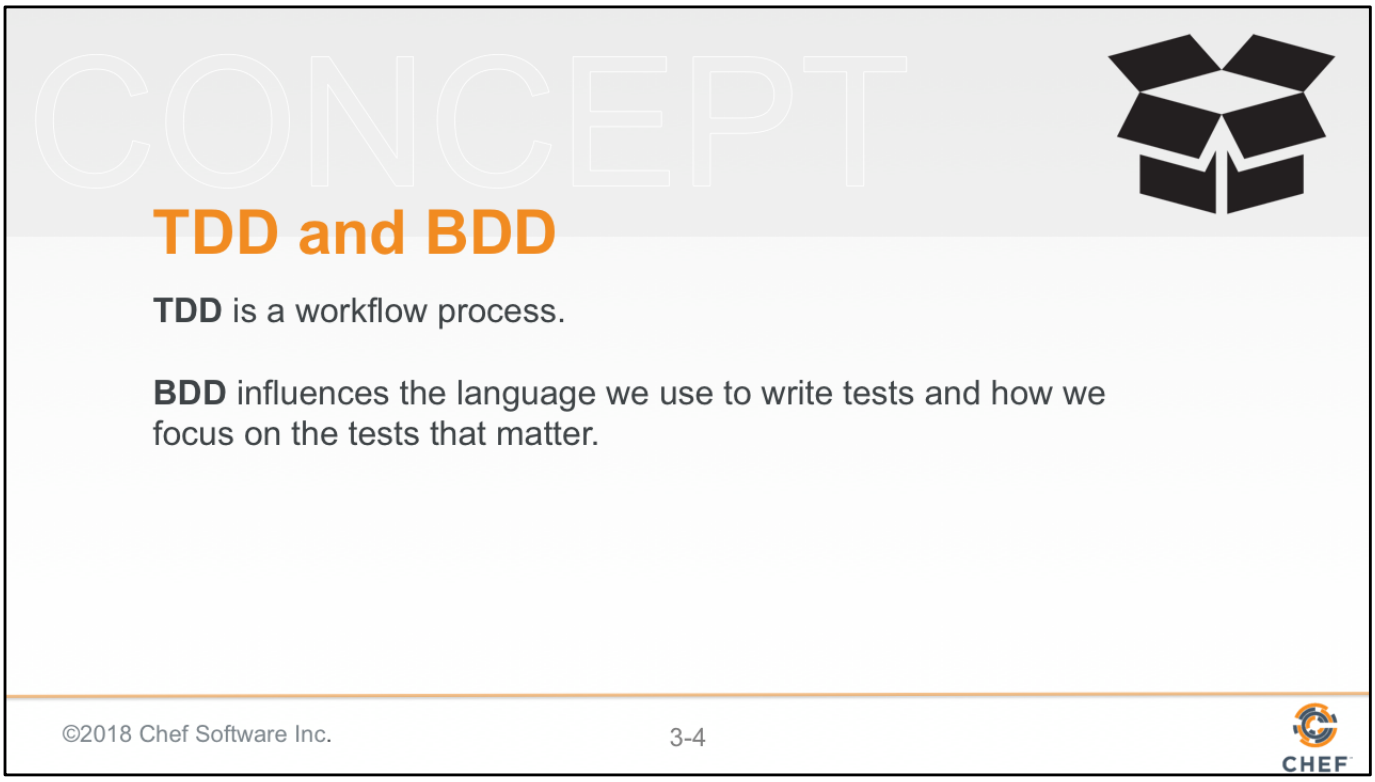
TDD generically focuses on the unit of software at any level. It is the process of writing the test first, implementing the unit, and then verifying the implementation with the test that was written.

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.



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.



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.

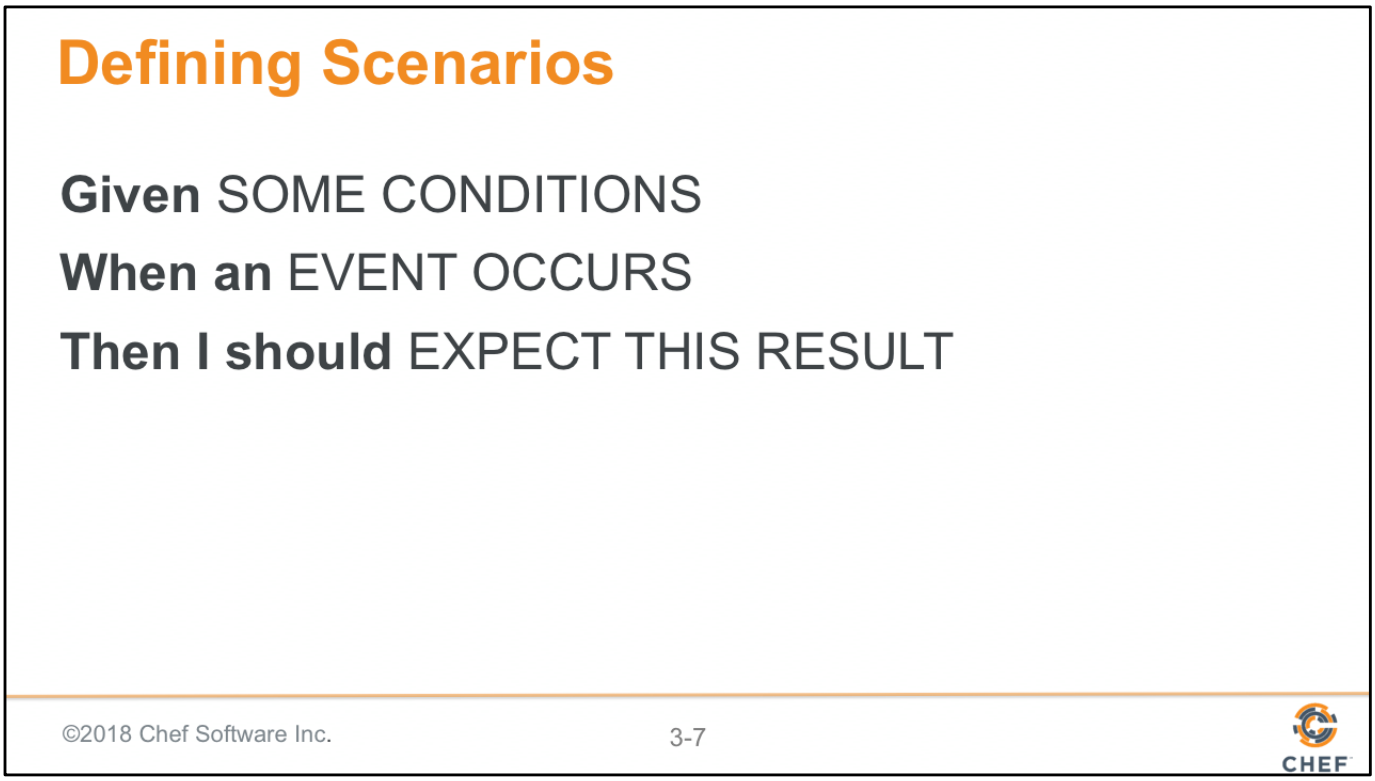
Building a Web Server

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.



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:

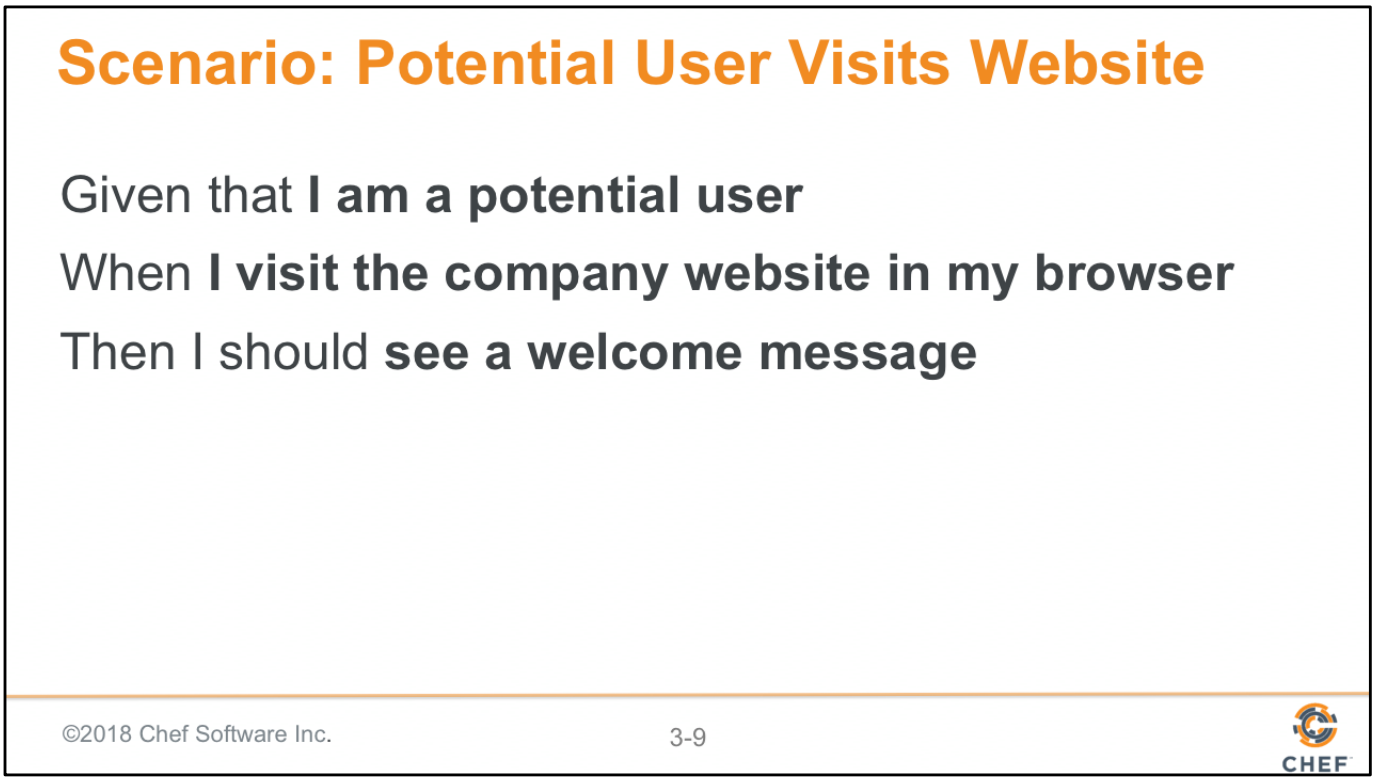
- Protect revenue
- Increase revenue
- Manage cost

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.

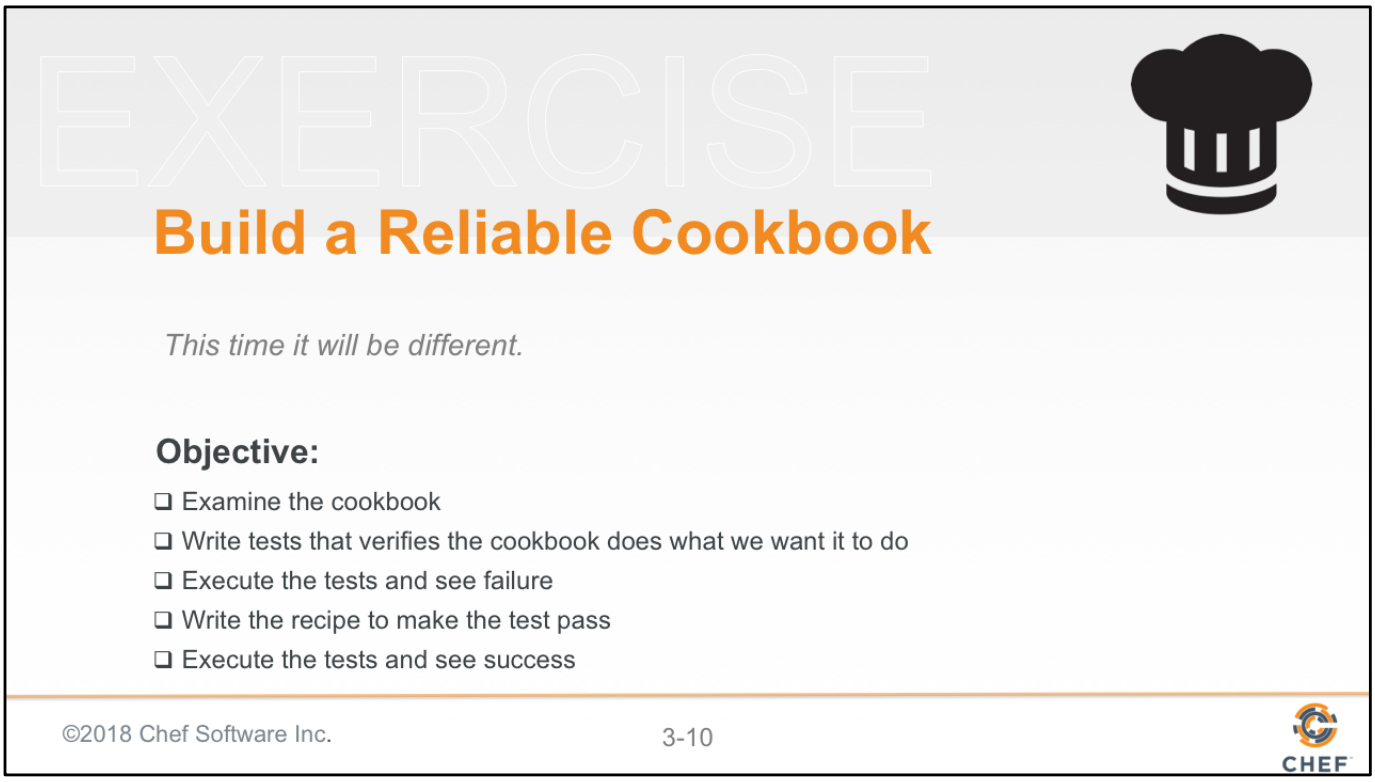


The typical reason for setting up a website is to allow customers, users, potential users to learn more about the company. The needs of the website may change in the future but the first minimum viable product (MVP) is to simply give our users the ability to find out more information.

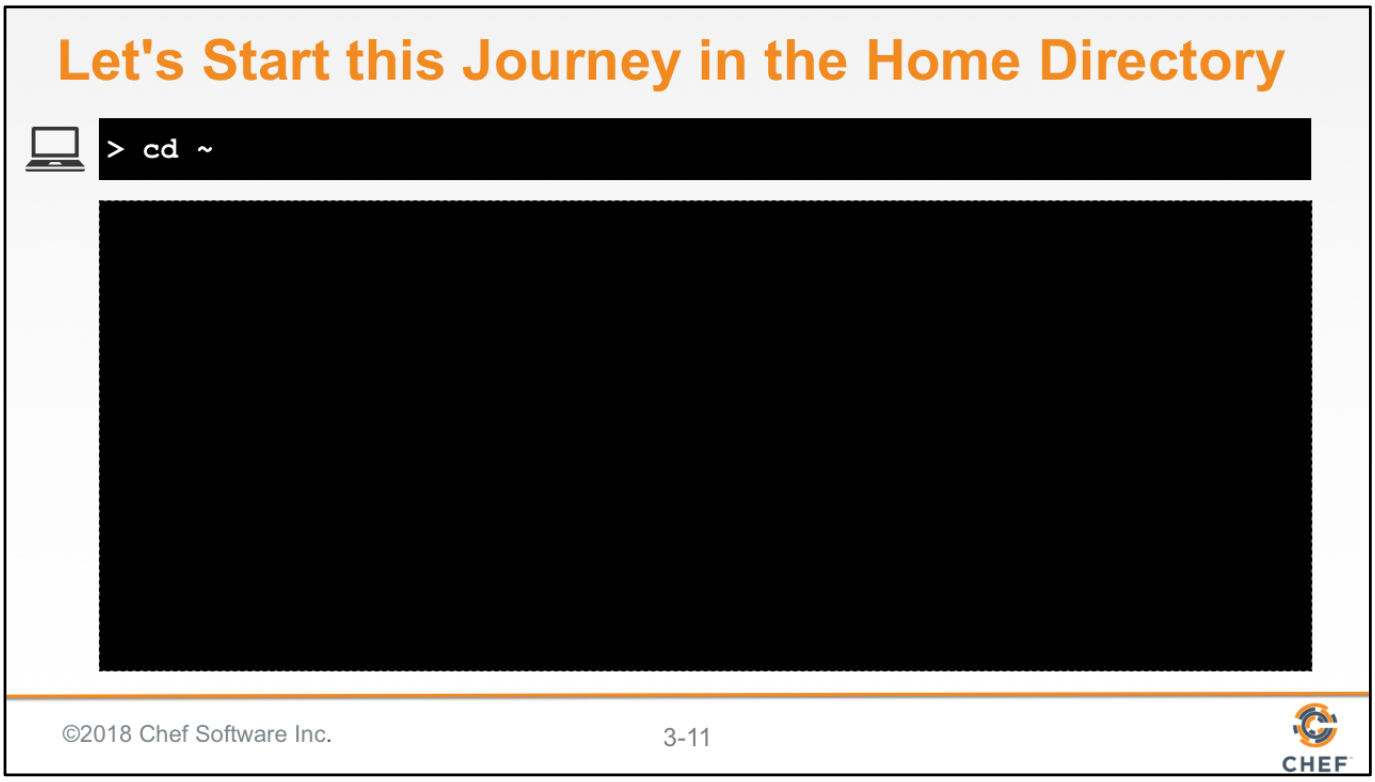
Our goal now is to define a scenario with this understanding.

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.



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.



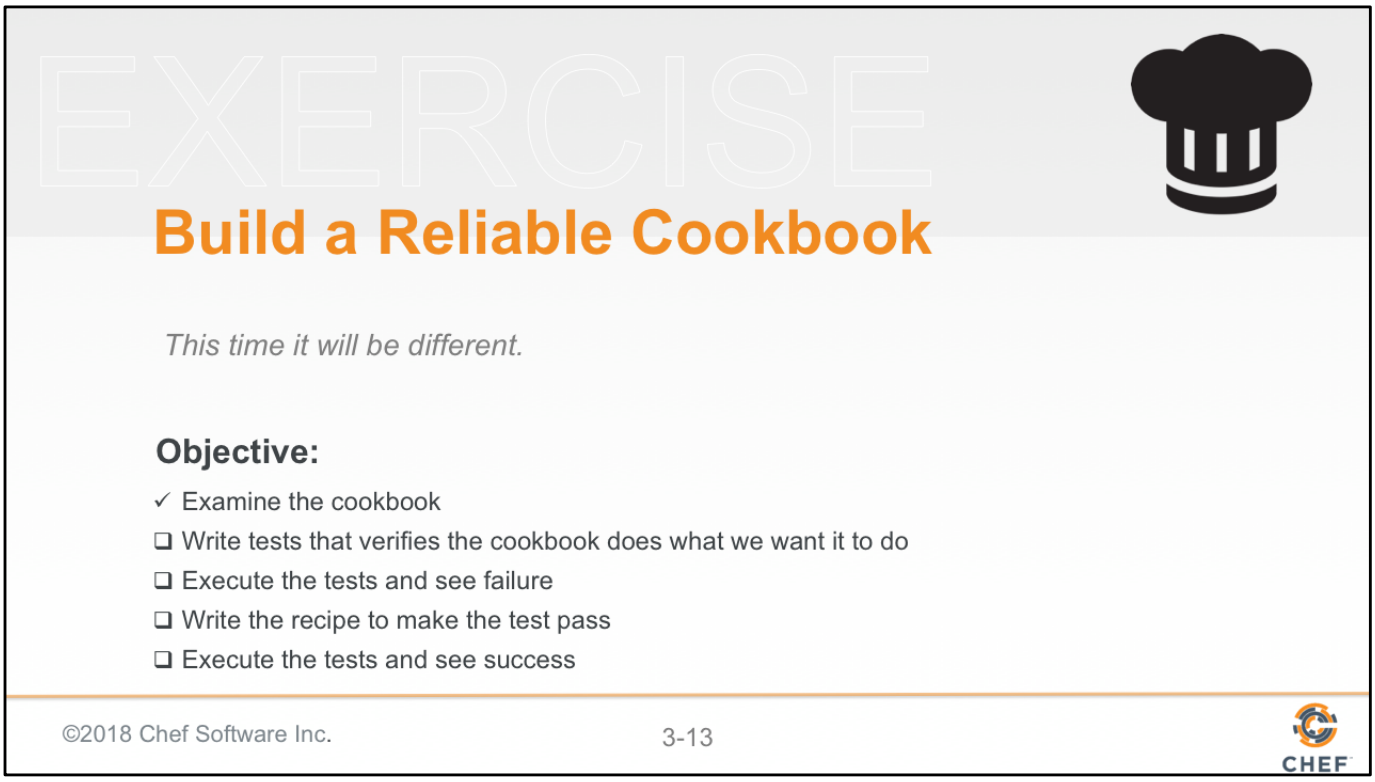
Let's start the journey on your workstation. From the home directory we are going to creating this cookbook.

View the Tests in the Generated Cookbook

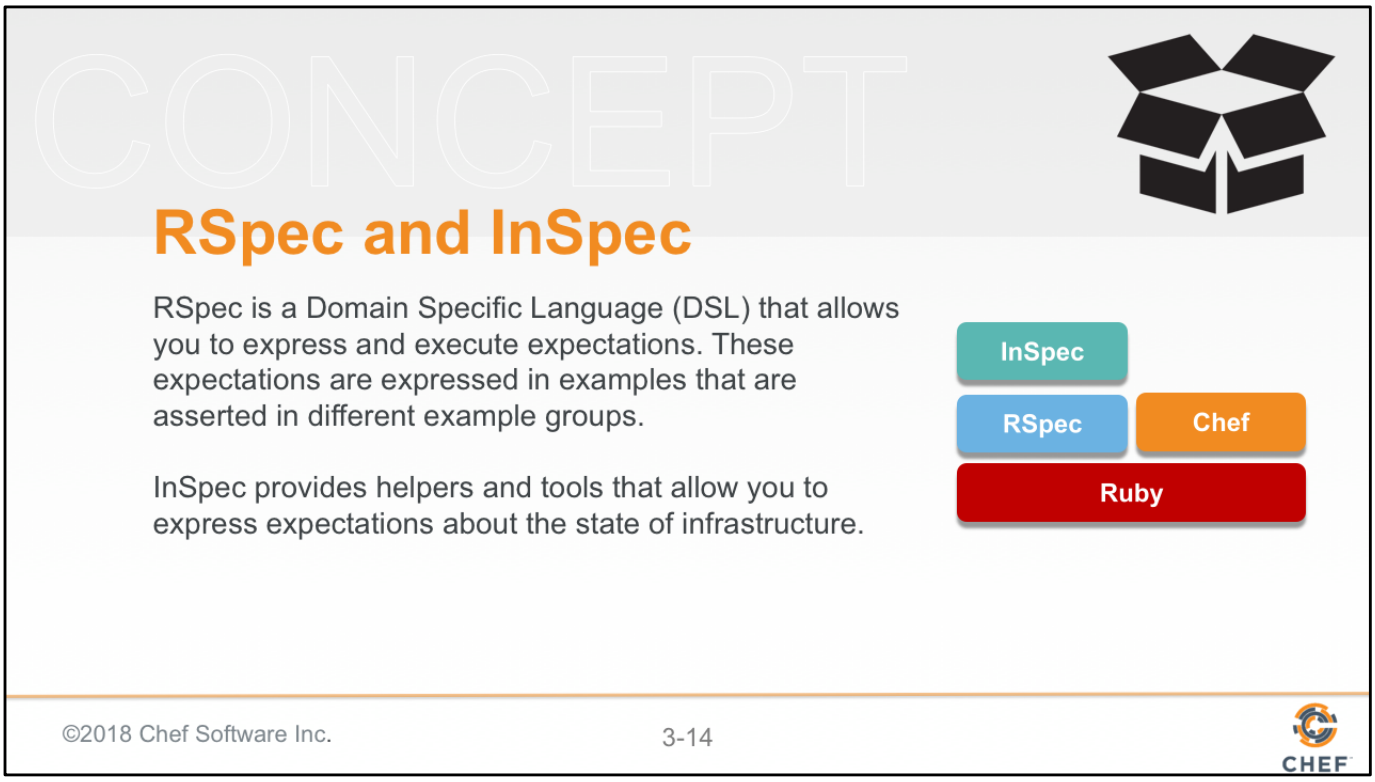
```
> tree apache
```

```
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.



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



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.

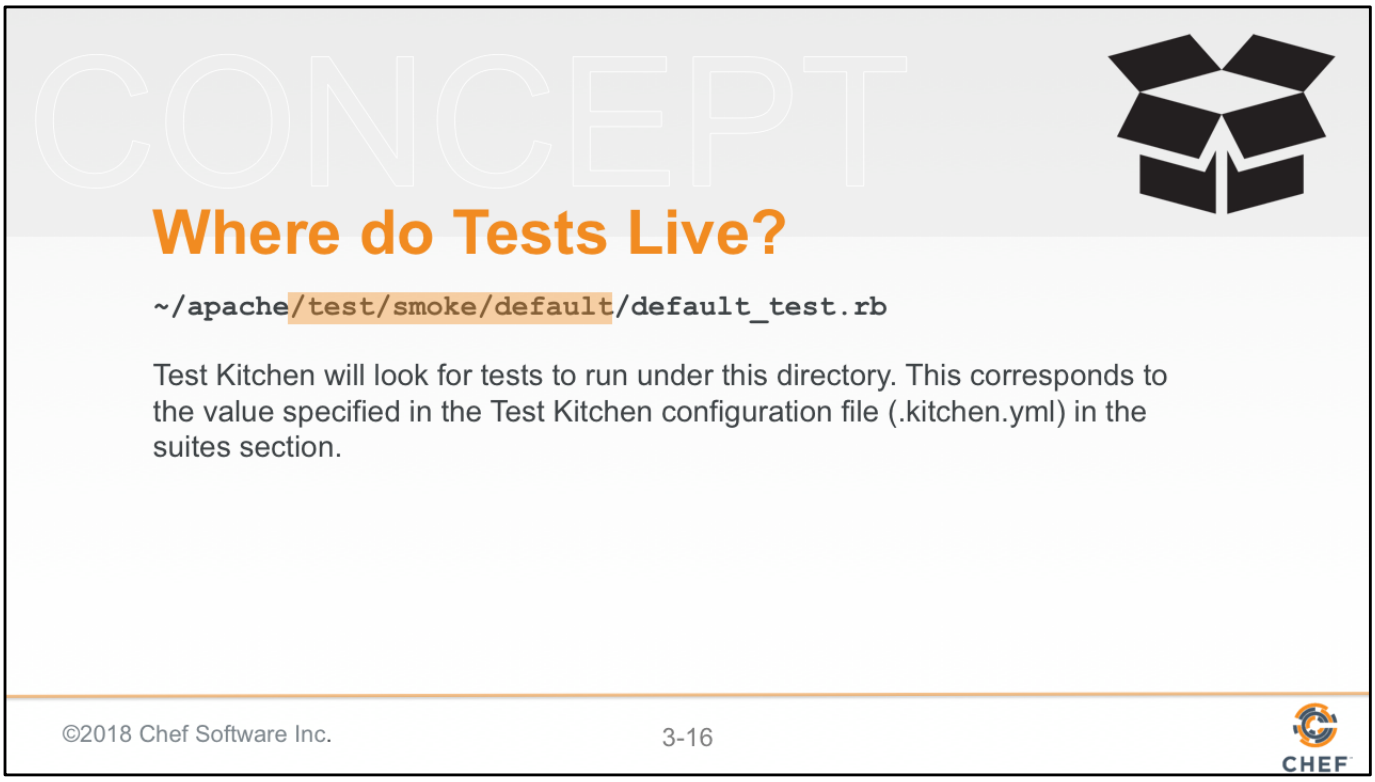
RSpec by itself grants us the framework, language, and tools. InSpec provides the knowledge about expressing expectations about the state of infrastructure.

Auto-generated Spec File in Cookbook

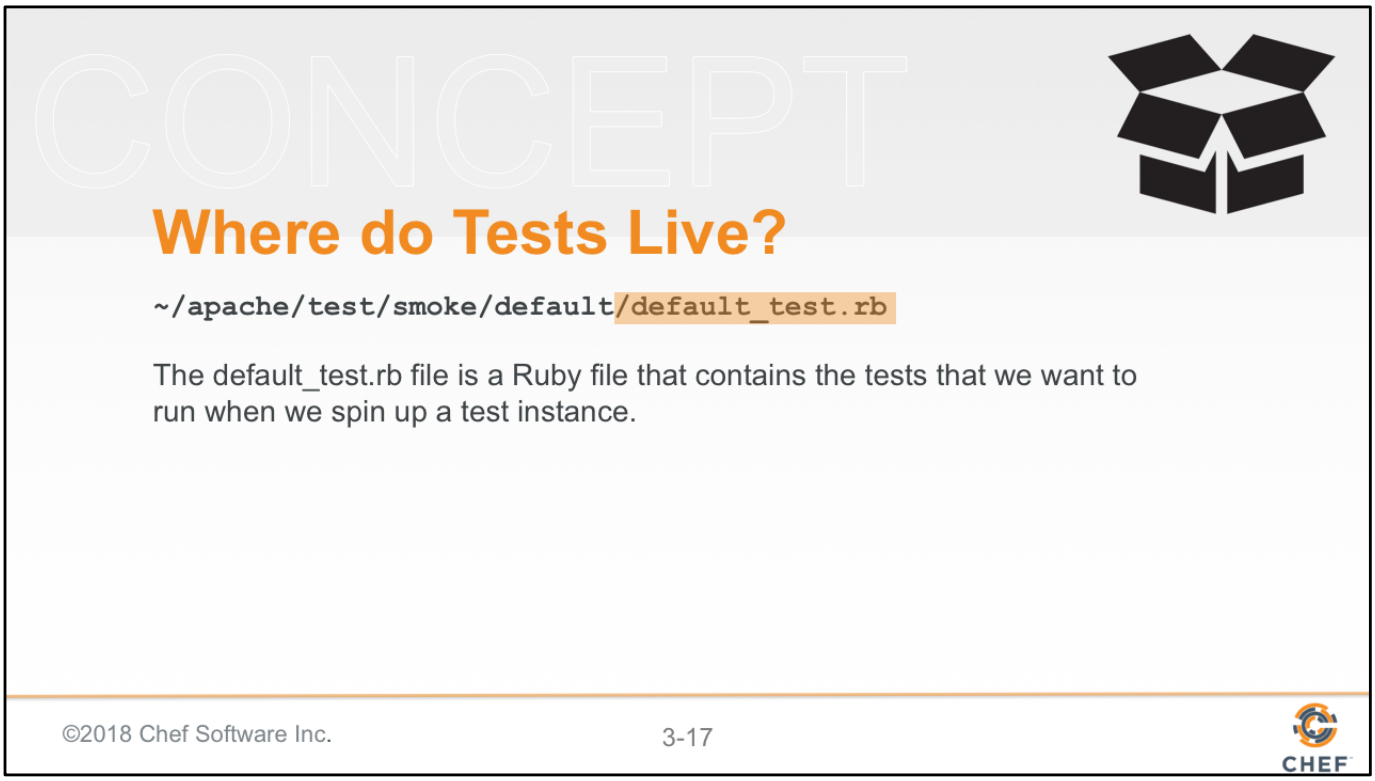
~/apache/test/smoke/default/default_test.rb

```
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.

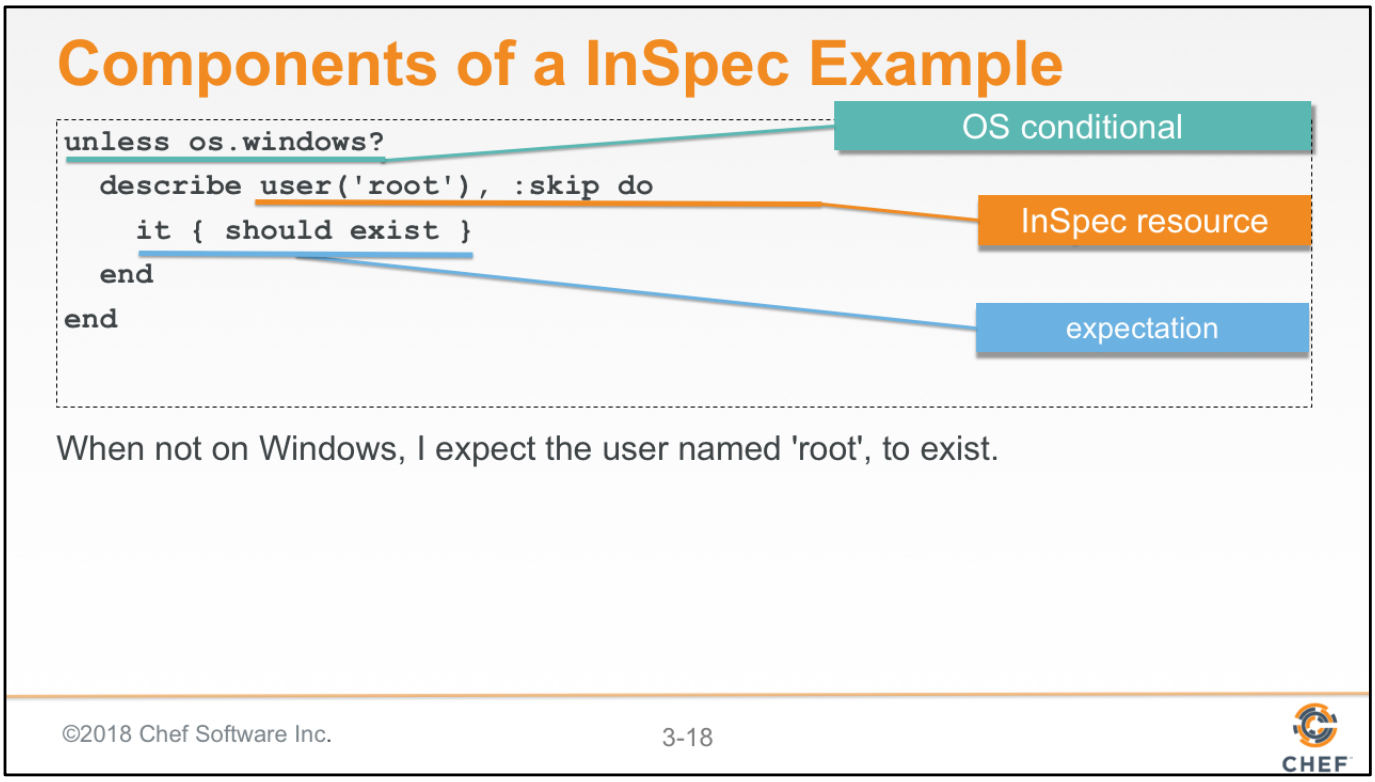


Let's take a moment to describe the reason behind this directory path. Within our cookbook we define a test directory and within that test directory we define another directory named smoke'. This is the basic file path that Test Kitchen expects to find the specifications defined in InSpec. The next part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



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.

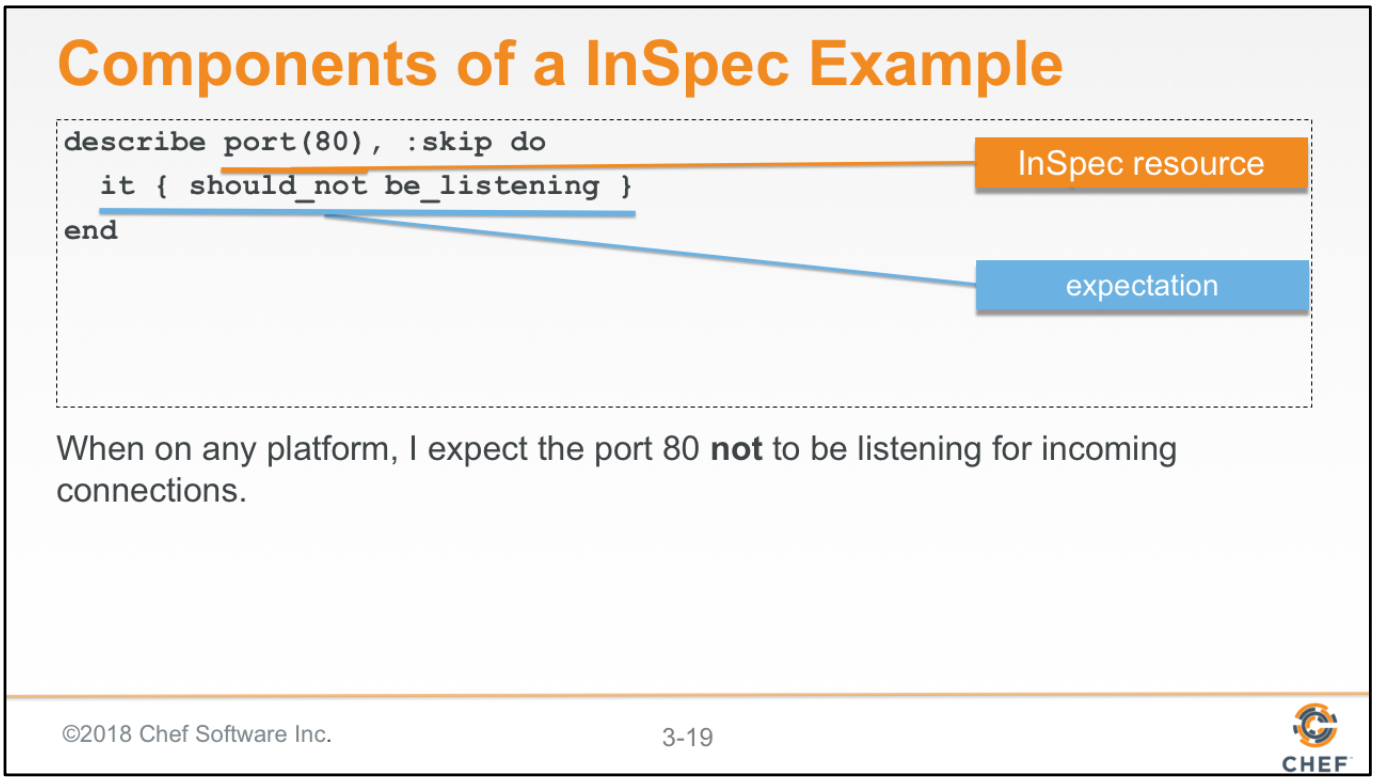
Let's open this default_test.rb file and review the contents of it.



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.



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.

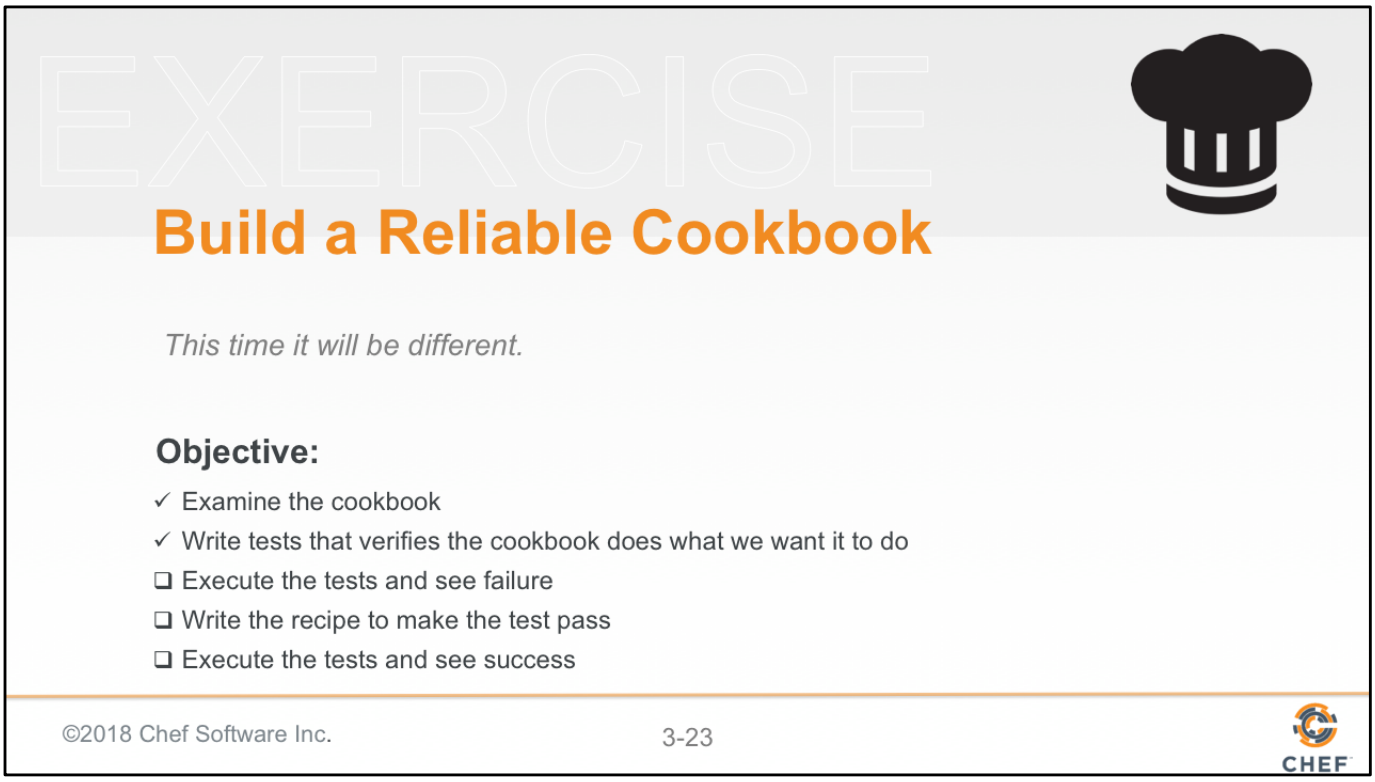
Add a Test to Validate a Working Website

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

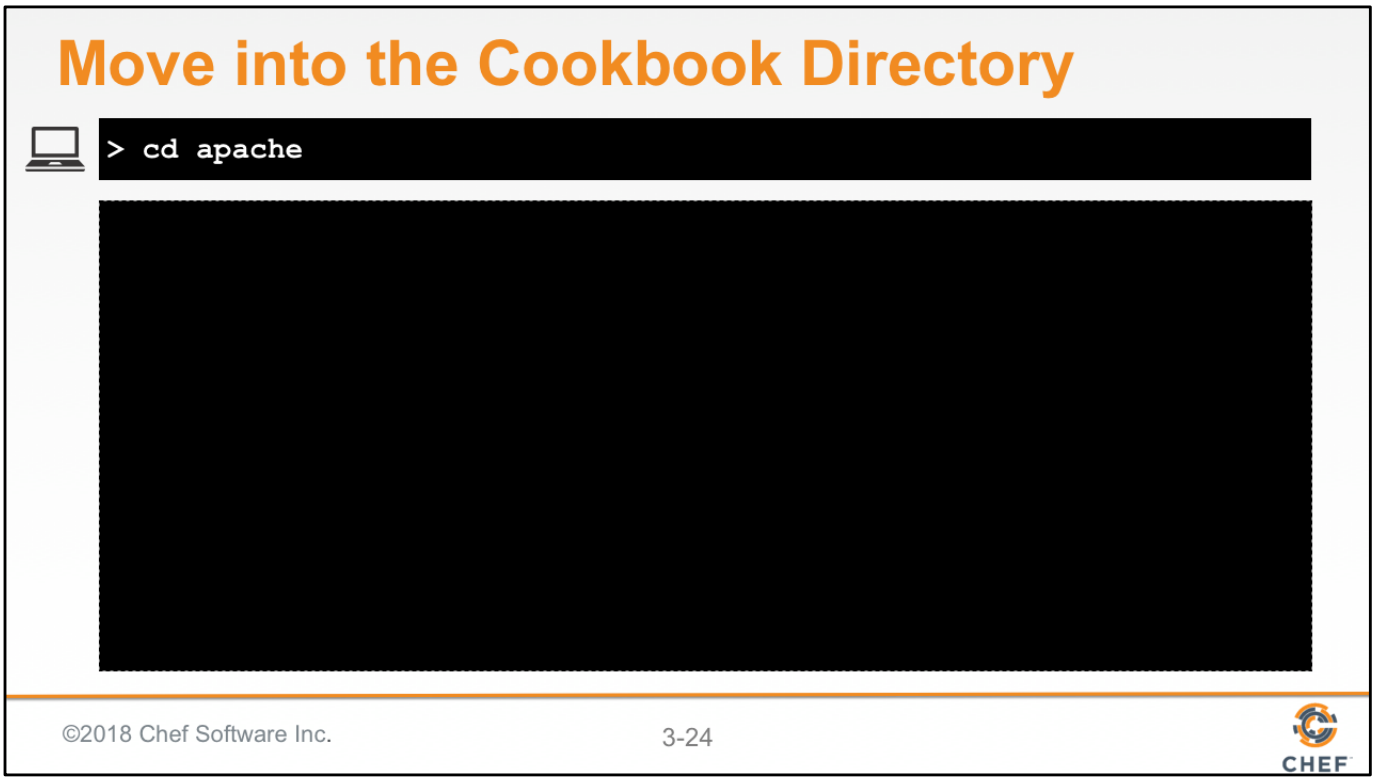
```
describe port(80) do
  it { should be_listening }
end

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'.



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



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

CHEF

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.

CHEF

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.

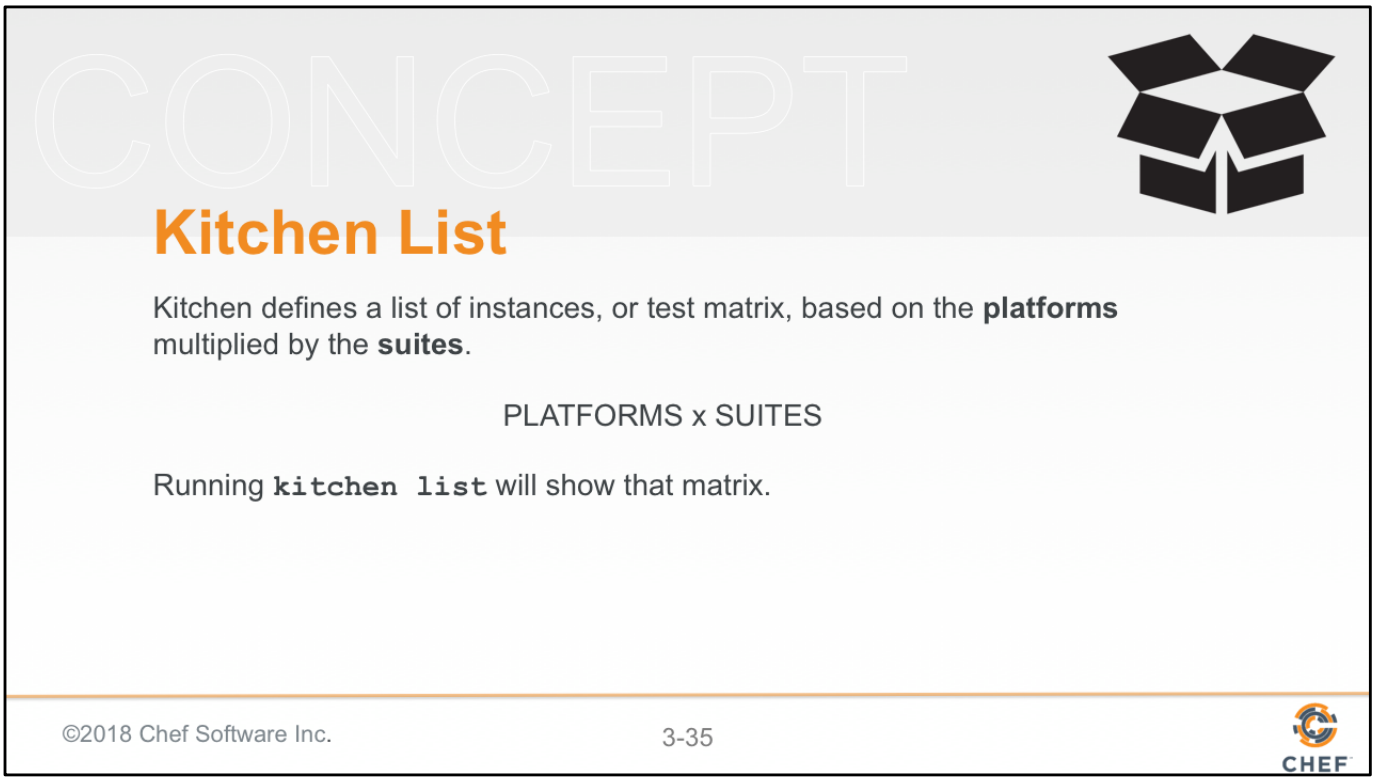
Add Settings to the Kitchen Configuration

~/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.

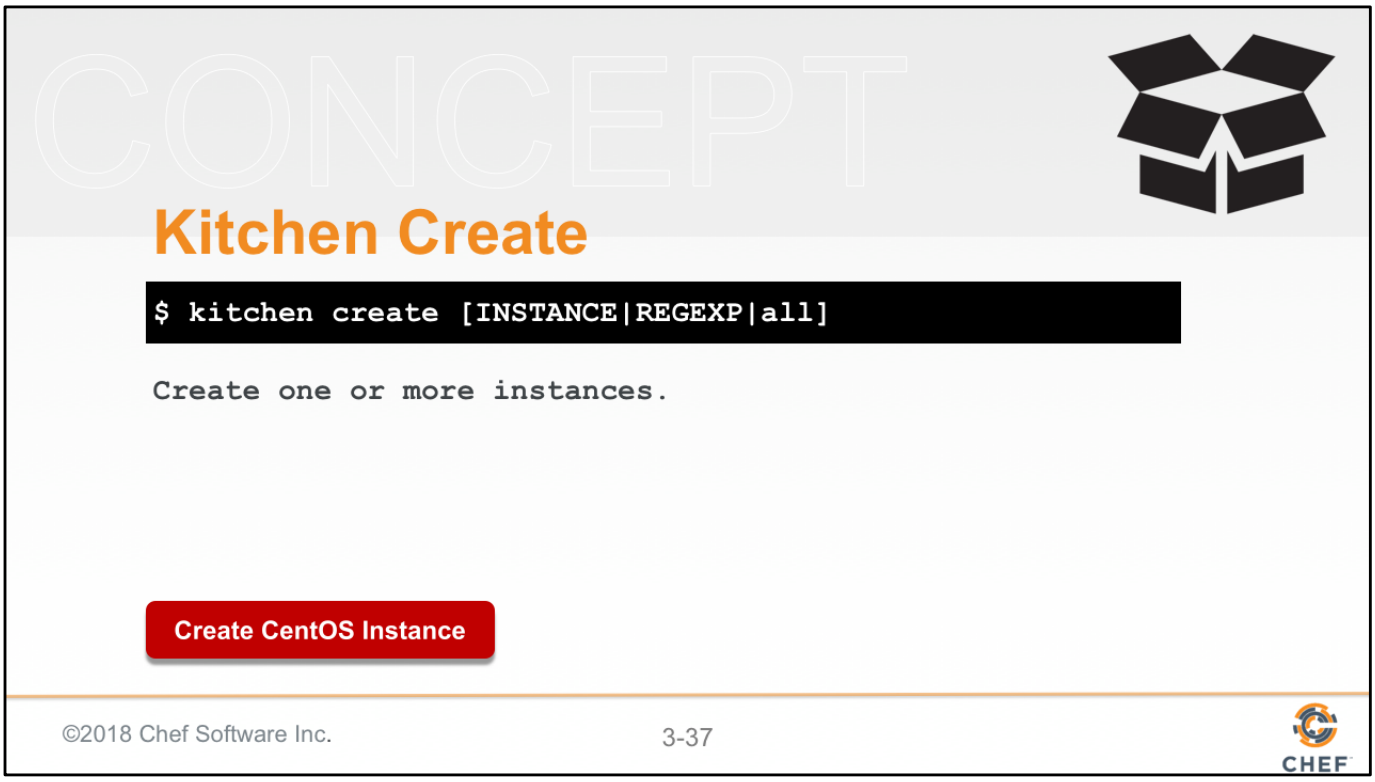


It is important to recognize that within the .kitchen.yml file we defined two fields that create a test matrix; the number of platforms we want to support multiplied by the number of test suites that we defined.

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.



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 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.

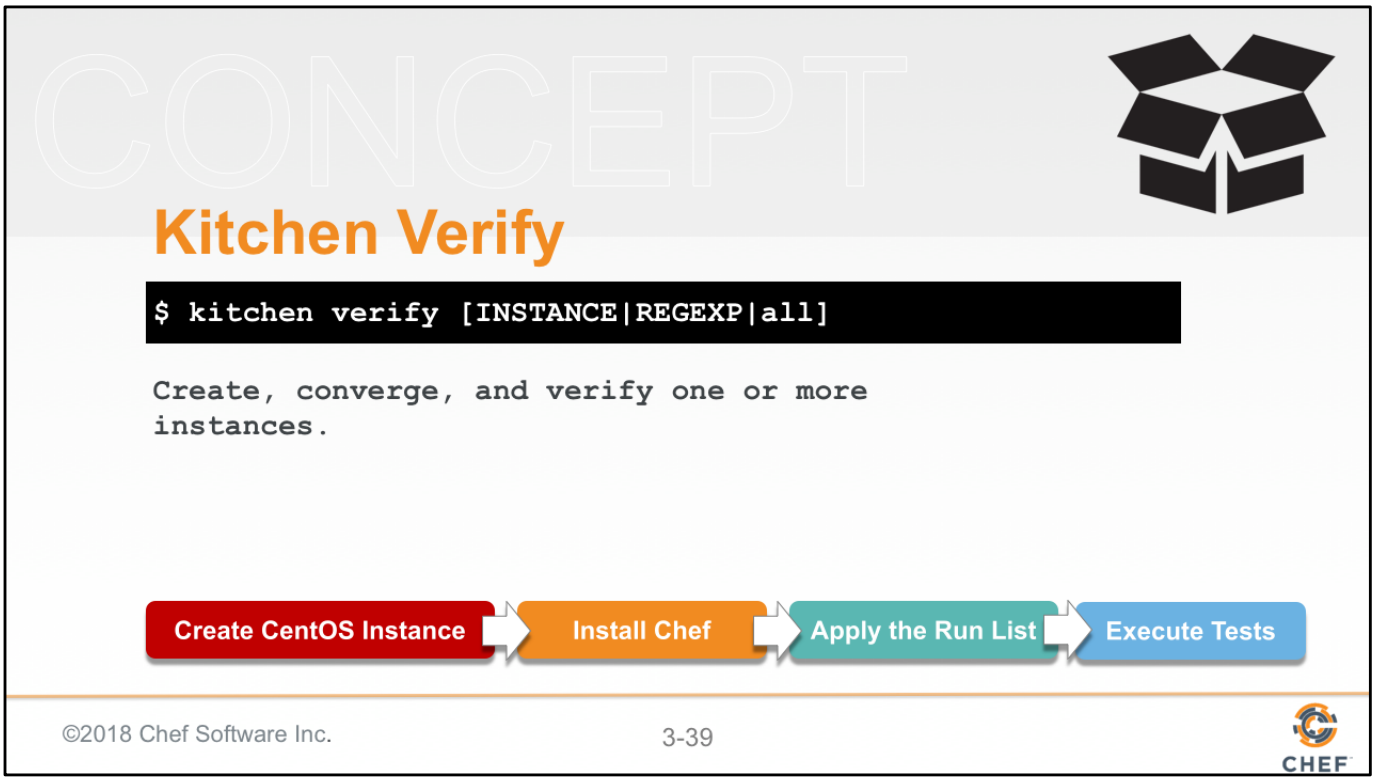
In our case, this command would use the Docker driver to create a docker image based on centos-6.9.

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.

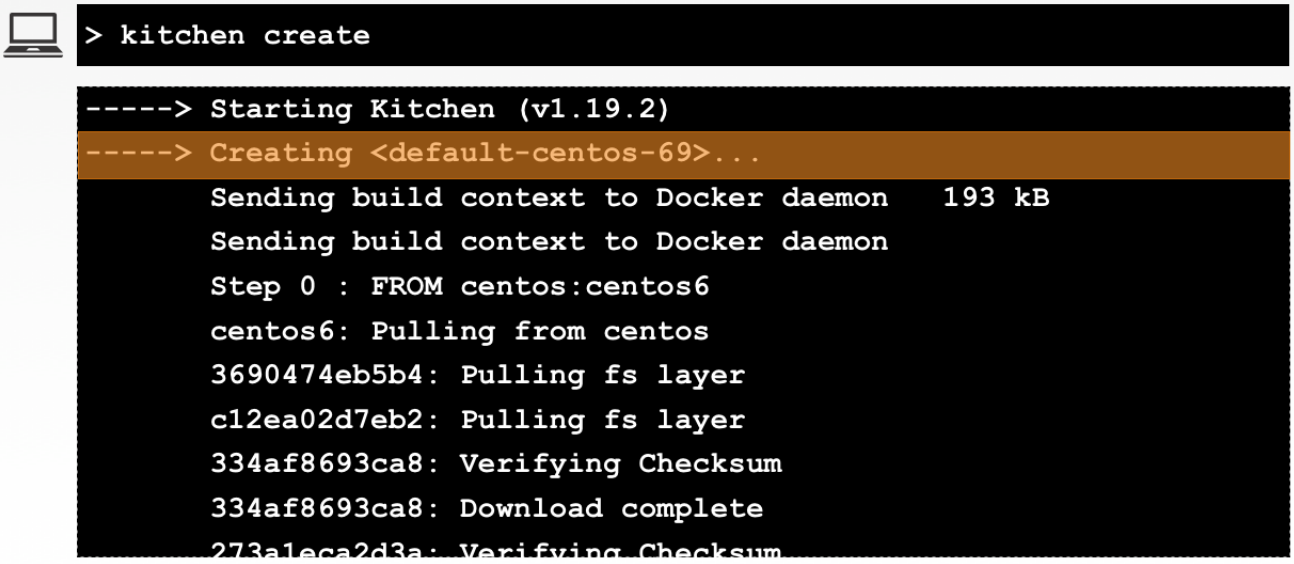
In our case, this command would take our image and install chef and apply the apache cookbook's default recipe.



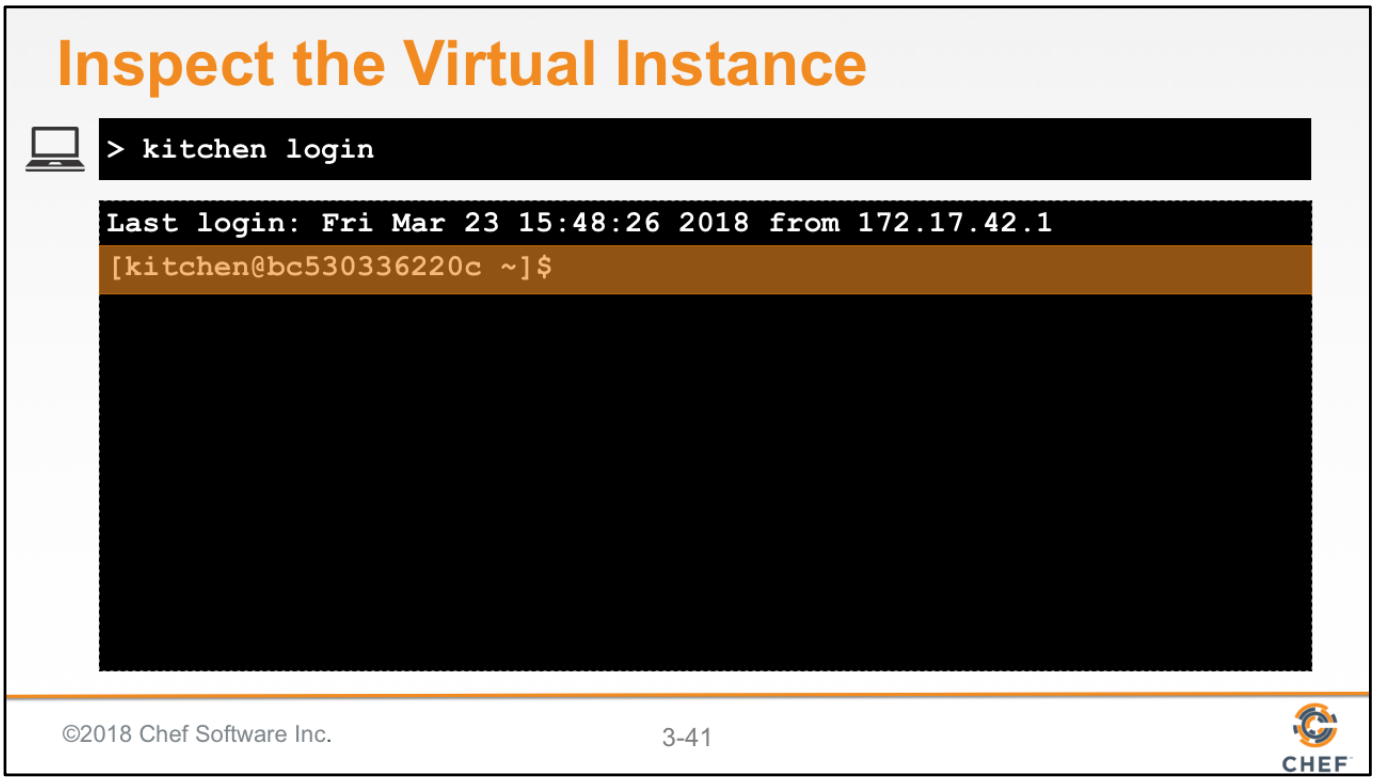
To verify an instance means to:

- 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

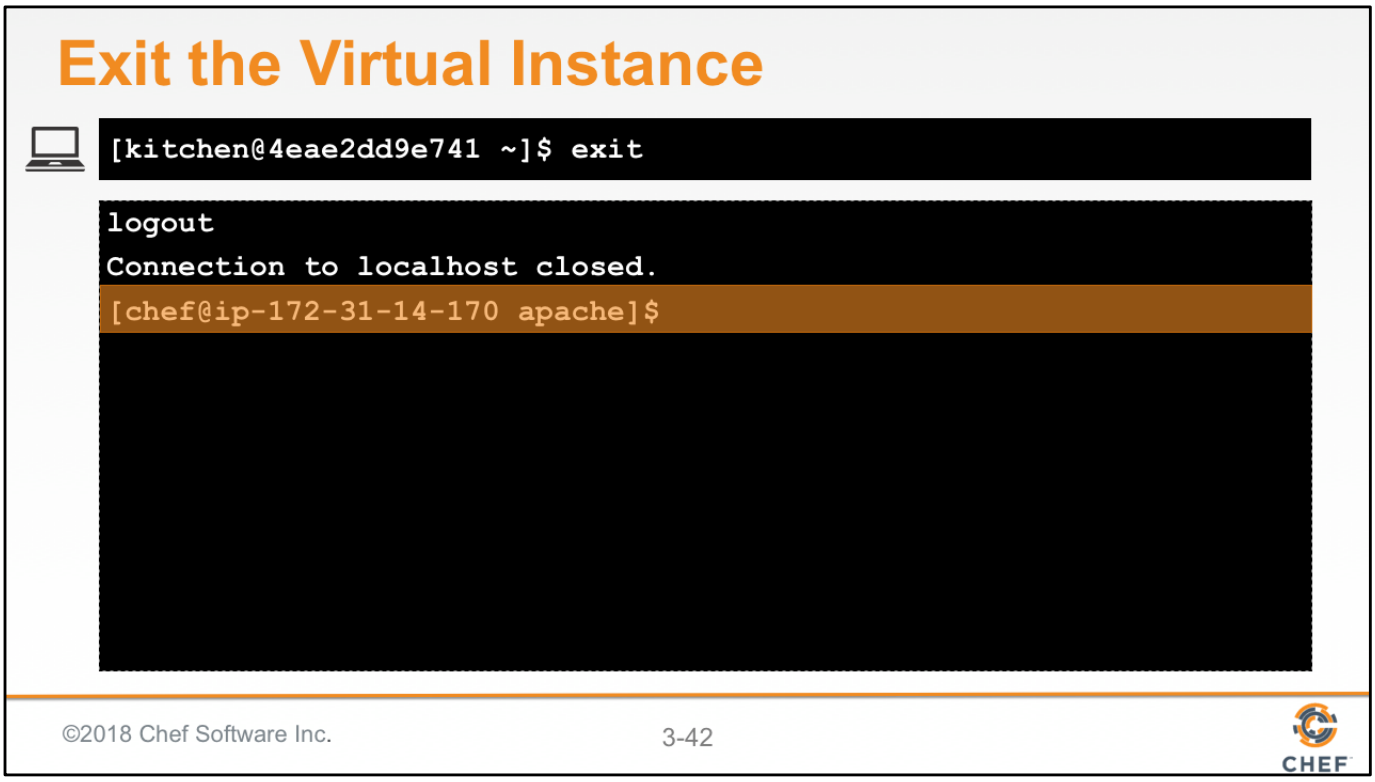


Create the instance with the following command. Here Test Kitchen will ask the driver specified in the kitchen configuration file to provision an instance for us.



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 in now logged into a virtual instance on a virtual instance.



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.

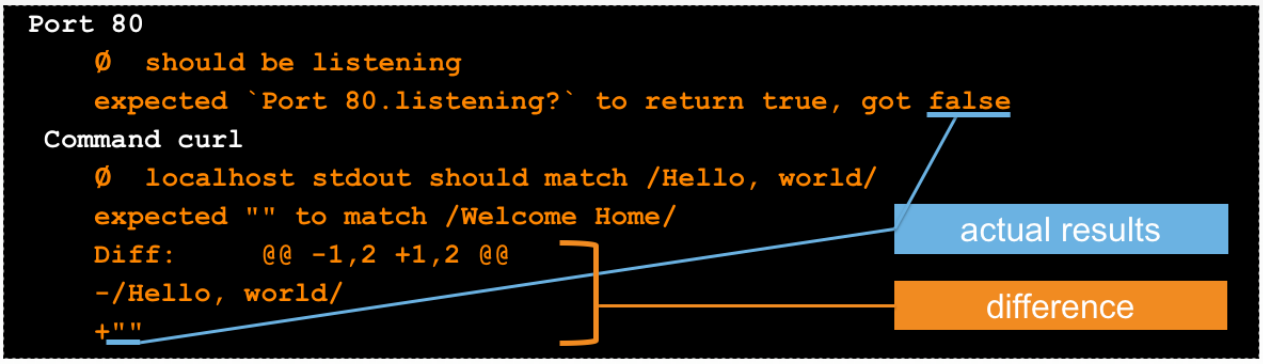
Something that is important to mention is that we could have simply run this command from the start. When no previous instance exists, no instance has been created or converged, this command will automatically perform those two steps. When the instance is running, however, the verification step is only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



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.

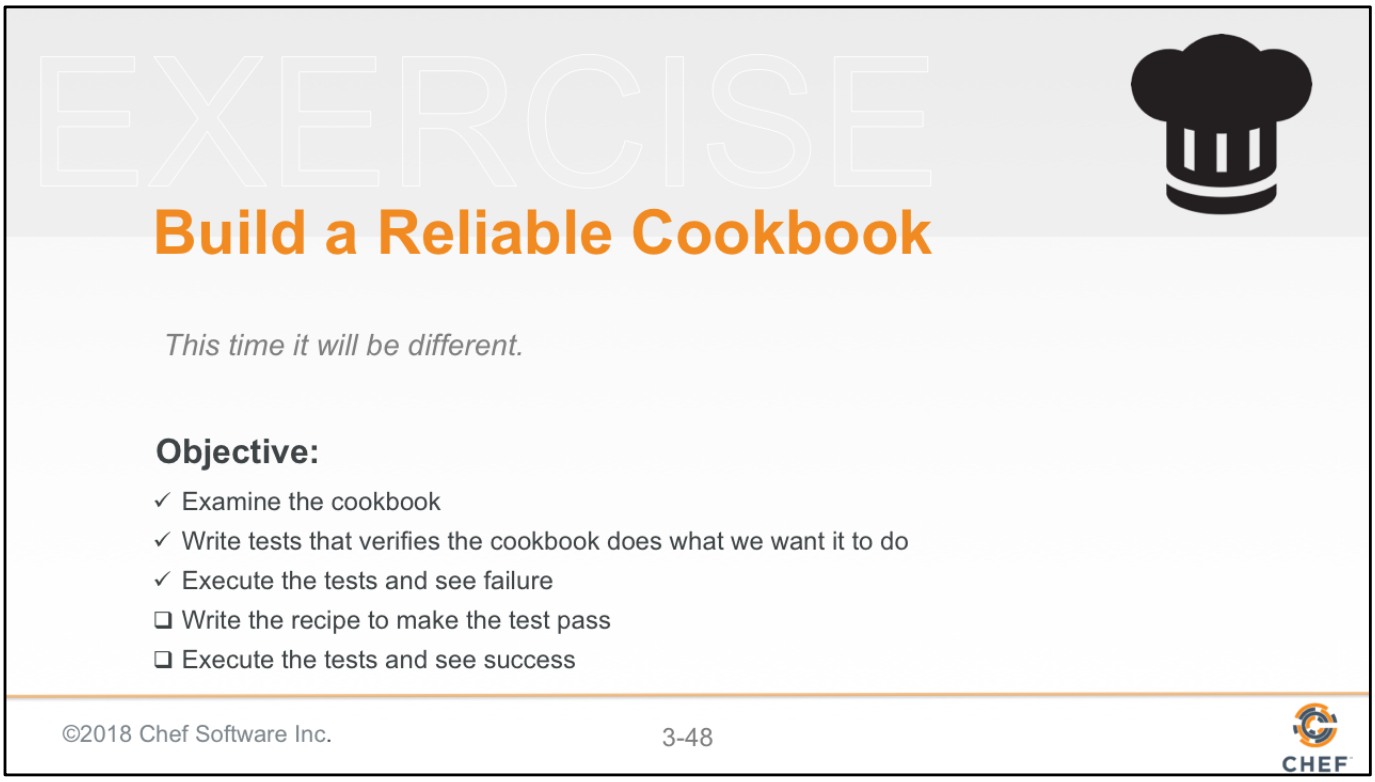
Examine the Test Summary

```
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.

CHEF

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.



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!'.

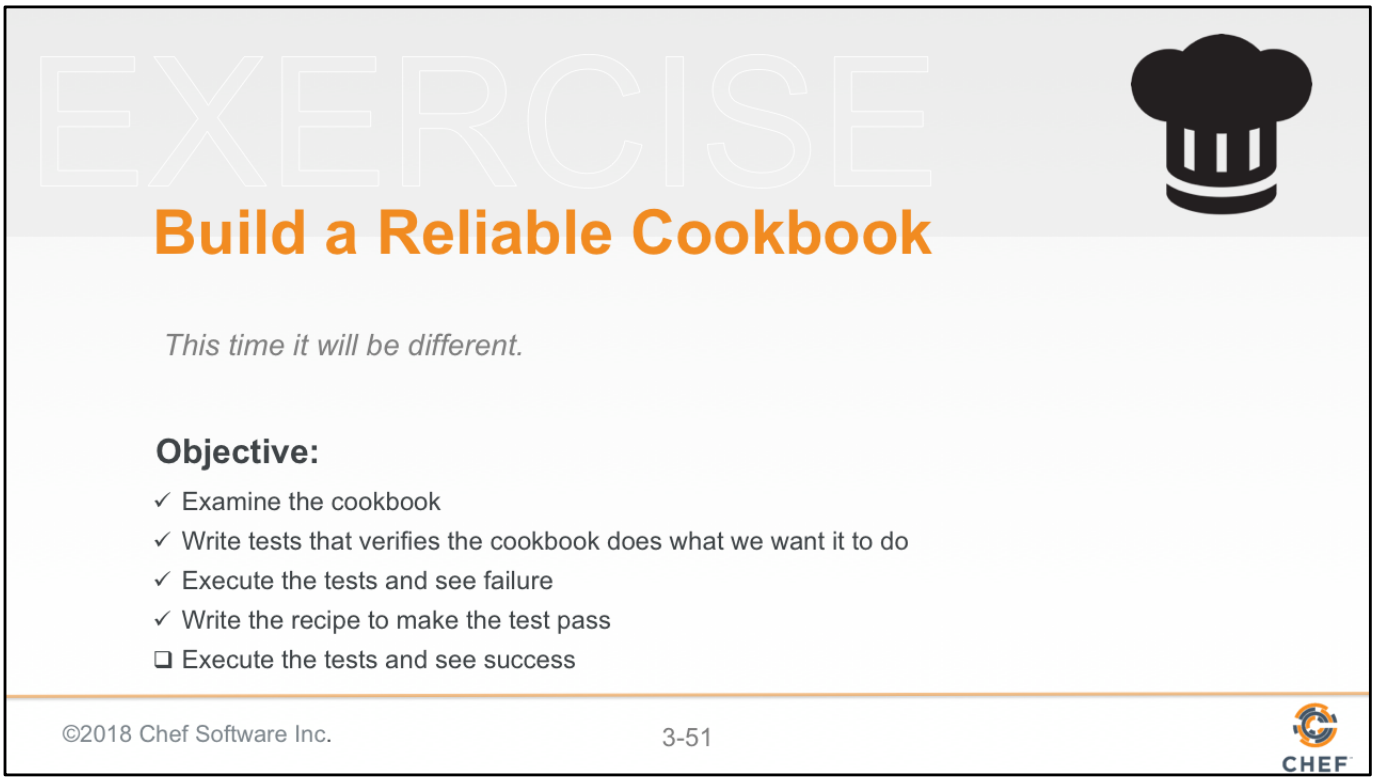
The package will install all the necessary software on the operating system. The file will create an HTML file with the desired content at a location pre-defined by the web server. The service resource will start the web server and then ensure that if we reboot the system the web server will start up.

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.



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

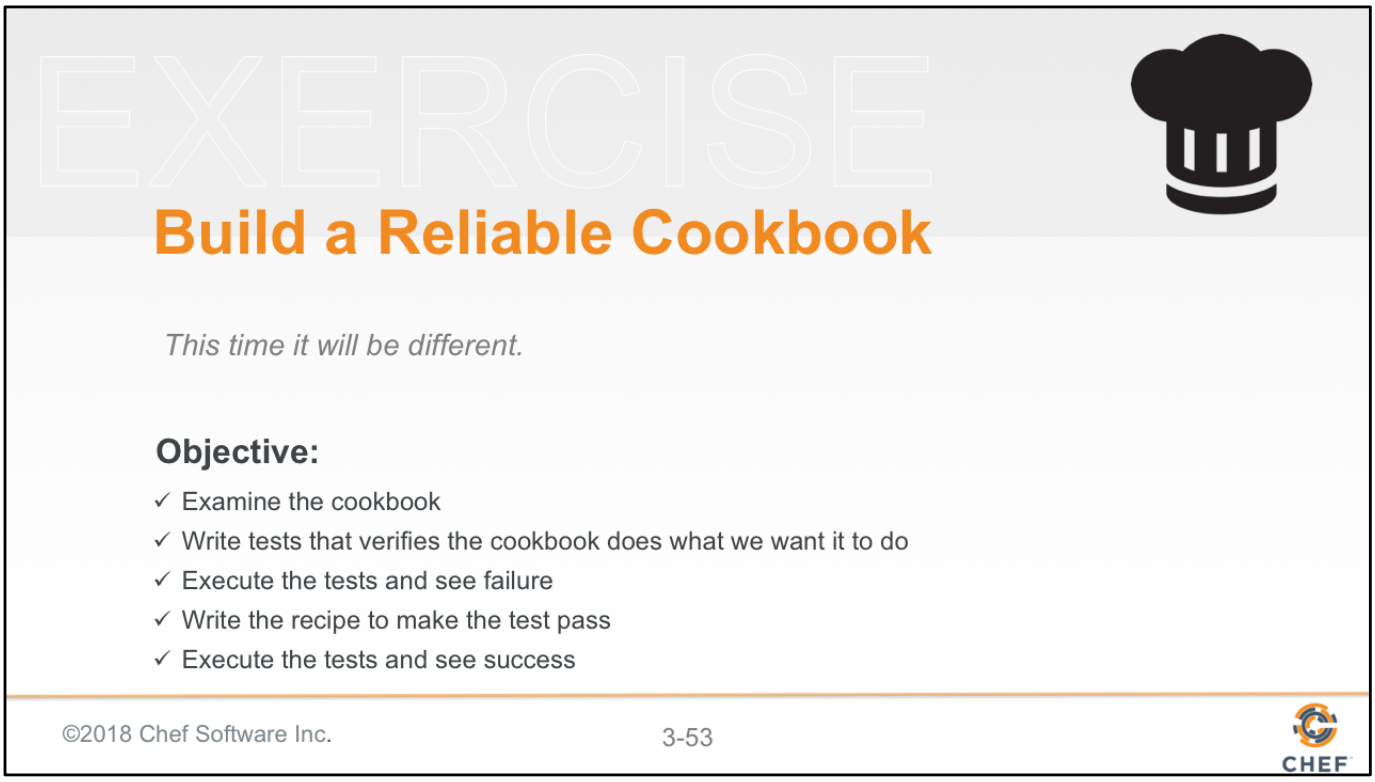
```
> kitchen verify
```

```
-----> 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
```


CHEF

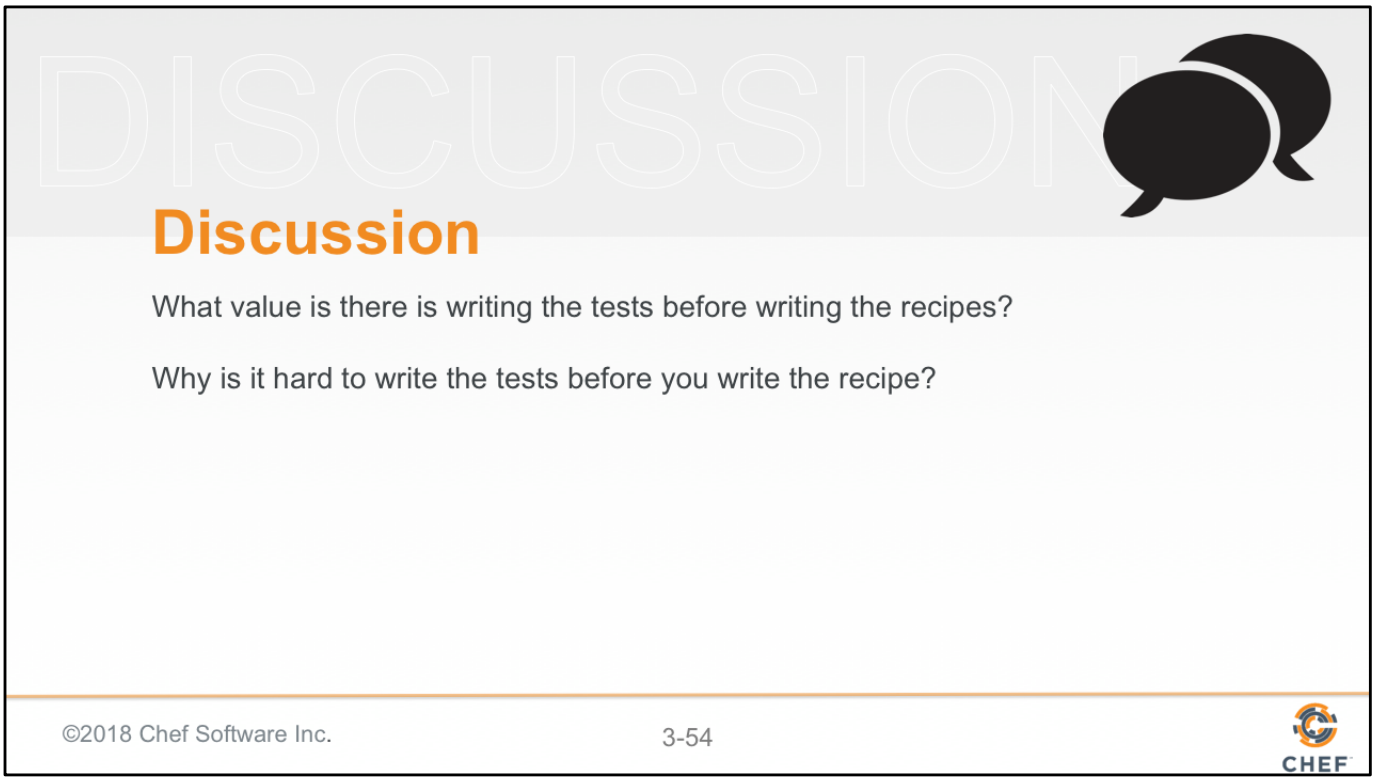
To verify the state of the virtual instance you run the 'kitchen verify' command. In the summary you should find the failing expectation no longer fails.

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.



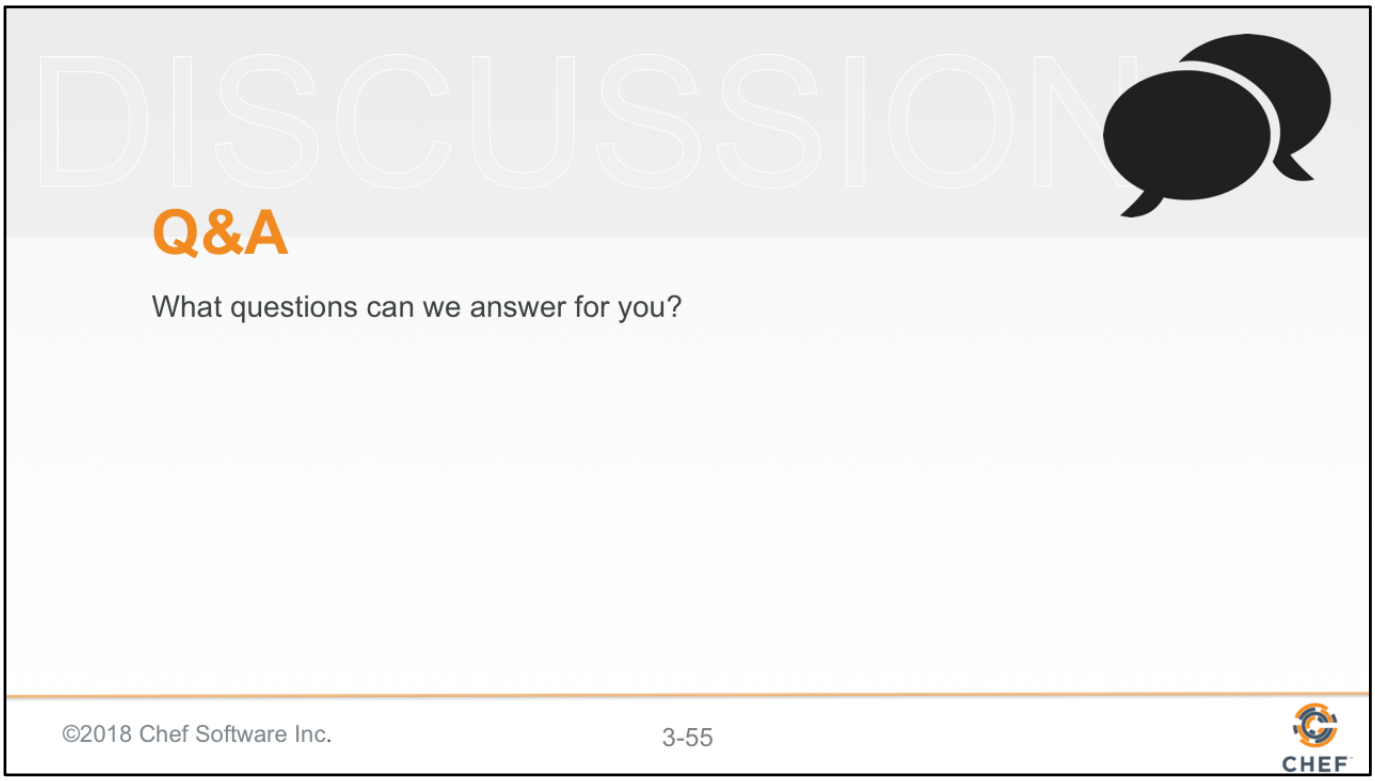
So you've done it. You have done Test Driven Development (TDD). Wrote a test. Saw it fail. Wrote a unit of code. Saw it pass.

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.

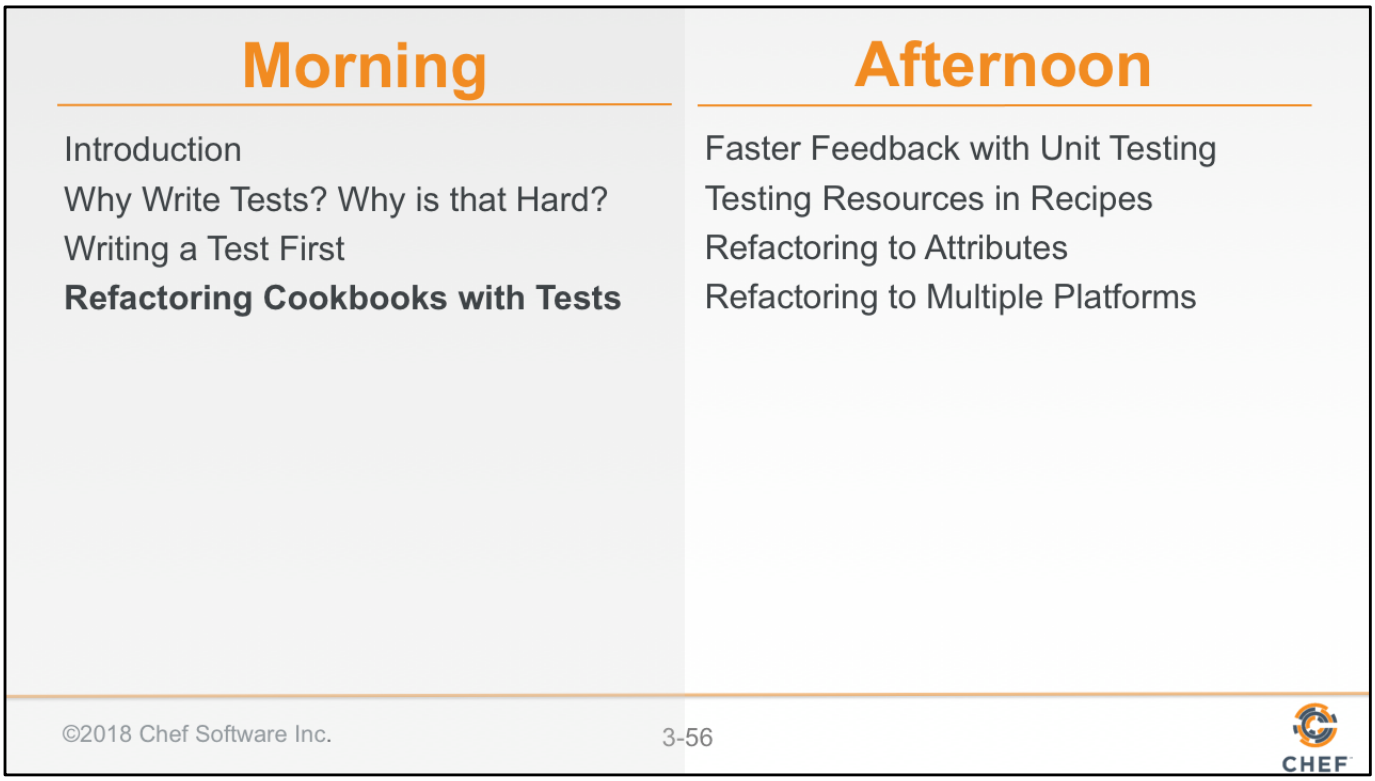


Now that you participated in writing a test and then the recipe let's have a discussion.

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.



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



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.


CHEF™