Local Cookbook Development

Course Introduction

Why would you want the Local Cookbook Development Badge?

- Earning this badge demonstrates your ability to develop a basic Chef cookbook

- By the end of the course, you should have some understanding of:
  - How to fully validate your cookbook's functionality before running it on real infrastructure
  - Cookbook components and ChefDK tools
  - How use Chef to locally
    - Interact with cookbooks
    - Compose recipes
    - Perform testing
    - Verifying intent

Meet your Course Author

Matt Saliano

Syllabus

- Introduction
- Setting Up the Environment
- Local Cookbook Development Basics
- Cookbook Components
- Design Patterns and Theory

Taking Advantage of Supplemental Course Material

- If you just watch the video lessons, you won't be prepared for the exam just yet
  - You will need to use your knowledge of the course concepts and apply it for the performance-based part of the exam
  - To fully prepare:
    - Make sure you do the course exercises
      - Understand the solution and be able to execute the solution without looking up the videos
      - Use your Linux Academy Server Labs to run through the exercises
    - Memorize the Note Cards that come with the course
      - Be sure to add your own note cards as you go through the lessons
    - Get perfect scores on the practice exam
  - You can also review the slides by going to course "Downloads"
  - Consult the "Course Features" area on the course page to better familiarize yourself

Local Cookbook Development

About the Exam

The Chef Certification Program

- The Chef Local Cookbook Development Badge is part of the Chef Certification program

    - The Chef Certification program validates your automation skills via badges and certifications
        - Badges reflect your mastery of solving problems with automation
        - Badges combine into a particular Chef Certification

    - Chef believes in the DevOps principle of “Kaizen”
        - A Japanese term meaning “continuous improvement”

Applicable Chef Certifications

Certified Chef Developer

Certified Chef Architect

Applicable Chef Certifications

Certified Chef Developer

Certified Chef Architect

Applicable Chef Certifications

Certified Chef Developer

Certified Chef Architect

Local Cookbook Development Exam Breakdown

- The Chef Local Cookbook Development exam is broken up into two parts
  - Part 1: Multiple Choice Questions
    - 40 questions
    - 1 hour to complete
    - Must pass with 80% or higher
    - Part 2 can only be scheduled after passing
      - This means you will not be able to take both parts of the exam on the same day
  - Part 2: Performance-based Scenario
    - 1 hour to complete
    - You will be provided a remote workstation that has the ChefDK installed
      - Includes text editors (examples: Atom, Vim)
      - Includes browser remote browser tabs (examples: Chef docs, objectives list)
    - You will need to work through a collection of tasks, for example:
      - Making tests pass
      - Authoring cookbooks

Exam Frequently Asked Questions

- What happens if I don't pass the exam?
  - The Chef Local Cookbook Development Exam is harder than Basic Chef Fluency Badge
    - If you don't pass one of the parts, you will need to wait two weeks before you can retake the exam
      - If you pass Part 1, you don't need to take it again when you retake the exam

- How do I register for an exam?
  - You need to create an account on https://training.chef.io and then you'll be able to purchase your exam voucher
  - The price is $175 for the Local Cookbook Development Badge exam as of February 2017

- Where do I go to see badges I've acquired?
  - You can see badges and certifications in your Chef Training account view

- How long is a certification valid for?
  - Certifications are valid for two years

Local Cookbook Development

Badge Overview

What are the high-level objectives of the Chef Local Cookbook Development Badge?

- Candidates must show that they have an understanding of:
  - Authoring cookbooks and setting up the local environment (and cookbook components)
  - ChefDK tools
  - Test Kitchen configuration
  - Available testing frameworks
  - Troubleshooting cookbooks
  - Search and data bags

- This course builds upon the Basic Chef Fluency Badge (BCFB) content
  - Be sure you've passed that exam before taking this course
  - This course will cover content that was in the BCFB course in more depth
  - This course will be much more hands-on

Exam Scope of Topics

- You can get a copy of the topics list from the Chef Training website: https://training.chef.io
  - The scope covered here is based on version 1.0.3 of the scope document
  - If you have this document handy, you can read along, but even listening prepares for what's ahead

- A candidate must be familiar with cookbook authoring and setup theory
  - Monolithic vs. single cookbook repository structure
    - Pros and cons of
      - A single repository per cookbook
      - An application repository
    - How the Chef workflow supports monolithic vs. single cookbooks
    - How to create a repository/workspace on the workstation

Exam Scope of Topics

- A candidate must be familiar with cookbook authoring and setup theory, cont.
  - Versioning of cookbooks
    - Why cookbooks should be versioned
    - The recommended methods of maintaining versions (e.g. knife/spork)
    - How to avoid overwriting cookbooks
    - Where to define a cookbook version
    - Semantic versioning
    - Freezing cookbooks
    - Re-uploading and freezing cookbooks
  - Structuring cookbook content
    - Modular content/reusability
    - Best practices around cookbooks that map 1:1 to a piece of software or functionality vs monolithic cookbooks
    - How to use common, core resources

Exam Scope of Topics

- A candidate must be familiar with cookbook authoring and setup theory, cont.
  - How metadata is used
    - How to manage dependencies
    - Cookbook dependency version syntax
    - What information to include in a cookbook (i.e. author, license, etc.)
    - Metadata settings
    - What 'suggests' in metadata means
    - What 'issues_url' in metadata means
  - Wrapper cookbook methods
    - How to consume other cookbooks in code via wrapper cookbooks
    - How to change cookbook behavior via wrapper cookbooks
    - Attribute value precedence
    - How to use the 'include_recipe' directive
    - What happens if the same recipe is included multiple times
    - How to use the 'depends' directive

Exam Scope of Topics

- A candidate must be familiar with cookbook authoring and setup theory, cont.
  - Using community cookbooks
    - How to use a public and private Supermarket
    - How to use community cookbooks
    - How to wrap community cookbooks
    - How to fork community cookbooks
    - How to use Berkshelf to download cookbooks
    - How to configure a Berksfile
    - How to use a Berksfile to manage a community cookbook and local cookbook with the same name
  - Using Chef resources vs. arbitrary commands
    - How to shell out to run commands
    - When/not to shell out
    - How to use the 'execute' resource
    - When/not to use the 'execute' resource
    - How to ensure idempotence

Exam Scope of Topics

- A candidate must be familiar with the ChefDK tools
  - 'chef' command
    - What the 'chef' command does
    - What 'chef generate' can create
    - How to customize content using 'generators'
    - The recommended way to create a template
    - How to add the same boilerplate text to every recipe created by a team
    - The 'chef gem' command
  - Foodcritic
    - What Foodcritic is
    - Why developers should lint their code
    - Foodcritic errors and how to fix them
    - Community coding rules and custom rules
    - Foodcritic commands
    - Foodcritic rules
    - How to exclude Foodcritic rules

Exam Scope of Topics

- A candidate must be familiar with the ChefDK tools, cont.
  - Berks
    - How to use Berks to work with upstream dependencies
    - How to work with GitHub and Supermarket
    - How to work with dependent cookbooks
    - How to troubleshoot Berks issues
    - How to lock cookbook versions
    - How to manage dependencies using Berks
    - Berks commands
  - Rubocop
    - How to use RuboCop to check Ruby styles
    - RuboCop vs. Foodcritic
    - RuboCop configuration and commands
    - Auto correction
    - How to be selective about the rules you run

Exam Scope of Topics

- A candidate must be familiar with the ChefDK tools, cont.
  - Test Kitchen
    - Writing tests to verify intent
    - How to focus tests on critical outcomes
    - How to test each resource component vs. how to test for desired outcomes
    - Regression testing

- A candidate must be familiar with Test Kitchen
  - Drivers
    - Test Kitchen provider and platform support
    - How to use .kitchen.yml to set up complex testing matrices
    - How to test a cookbook on multiple deployment scenarios
    - How to configure drivers

Exam Scope of Topics

- A candidate must be familiar with Test Kitchen, cont.
  - Provisioner
    - The available provisioners
    - How to configure provisioners
    - When to use chef-client vs. chef-solo vs. Chef
    - How to use the shell provisioner
  - Suites
    - What a suite is
    - How to use suites to test different recipes in different environments
    - Testing directory for InSpec
    - How to configure suites

Exam Scope of Topics

- A candidate must be familiar with Test Kitchen, cont.
  - Platforms
    - How to specify platforms
    - Common platforms
    - How to locate base images
    - Common images and custom images
  - Kitchen commands
    - The basic Test Kitchen workflow
    - 'kitchen' commands
    - When tests get run
    - How to install bussers
    - What 'kitchen init' does

Exam Scope of Topics

- A candidate must be familiar with cookbook components
  - Directory structure of a cookbook
    - What the components of a cookbook are
    - What siblings of cookbooks in a repository are
    - The default recipe and attribute files
    - Why there is a 'default' subdirectory under 'templates'
    - Where tests are stored
  - Attributes and how they work
    - What attributes are
    - Attributes as a nested hash
    - How attributes are defined
    - How attributes are named
    - How attributes are referenced
    - Attribute precedence levels
    - What Ohai is
    - What the 'platform' attribute is
    - How to use the 'platform' attribute in recipes

Exam Scope of Topics

- A candidate must be familiar with cookbook components, cont.
  - Files and templates – the difference, how they work, when to use each
    - How to instantiate files on nodes
    - The difference between 'file', 'cookbook_file', 'remote_file', and 'template'
    - How to write templates
    - What 'partial templates' are
    - Common file-related resource actions and properties
    - ERB syntax
  - Custom resources – How they are structured; where they go
    - What custom resources are
    - How to consume resources specified in another cookbook
    - Naming conventions
    - How to test custom resources

Exam Scope of Topics

- A candidate must be familiar with cookbook components, cont.
  - Libraries
    - What libraries are and when to use them
    - Where libraries are stored

- A candidate must be familiar with available testing frameworks
  - InSpec
    - How to test common resources with InSpec
    - InSpec syntax
    - How to write InSpec tests
    - How to run InSpec tests
    - Where InSpec tests are stored

Exam Scope of Topics

- A candidate must be familiar with available testing frameworks, cont.
  - ChefSpec
    - What ChefSpec is
    - The ChefSpec value proposition
    - What happens when you run ChefSpec
    - ChefSpec syntax
    - How to write ChefSpec tests
    - How to run ChefSpec tests
    - Where ChefSpec tests are stored
  - Generic testing topics
    - The test-driven development (TDD) workflow
    - Where tests are stored
    - How tests are organized in a cookbook
    - Naming conventions – How Test Kitchen finds tests
    - Tools to test code "at rest"
    - Integration testing tools
    - Tools to run code and test the output
    - When to use ChefSpec in the workflow
    - When to use Test Kitchen in the workflow
    - Testing intent
    - Functional vs. unit testing

Exam Scope of Topics

- A candidate must be familiar with troubleshooting
  - Reading test-kitchen output
    - Test Kitchen phases and associated output
  - Compile vs. Converge
    - What happens during the compile phase of a chef-client run
    - What happens during the converge phase of a chef-client run
    - When pure Ruby gets executed
    - When Chef code gets executed

Exam Scope of Topics

- A candidate must be familiar with search and data bags
  - Data bags
    - What data bags are
    - Where data bags are stored
    - When to use data bags
    - How to use data bags
    - How to create a data bag
    - How to update a data bag
    - How to search data bags
    - Chef Vault
    - The difference between data bags and attributes
    - What 'knife' commands to use to CRUD (Create, Read, Update, Delete) data bags

Exam Scope of Topics

- A candidate must be familiar with search and data bags, cont.
  - Search
    - What data is indexed and searchable
    - Why you would search in a recipe
    - Search criteria syntax
    - How to invoke a search from the command line
    - How to invoke a search from within a recipe

The Linux Academy Advantage

- At Linux Academy, we don't just teach to the test
    - We prefer to be as interactive as possible, allowing you to learn by sight, sound, and touch
    - We do this by giving you lessons, guides, and environments to get hands-on experience with
    - You won't just be able to succeed in the exam, but you'll be fully prepared to implement these skills
    - You'll be expected to complete all lessons, exercises, and the practice exam to demonstrate your understanding of the technology

Local Cookbook Development

Introduction to ChefDK

What is ChefDK?

- ChefDK is also known as the Chef Development Kit
  - ChefDK includes:
    - A built-in Ruby runtime
    - chef-client and ohai
    - Testing tools:
      - Test Kitchen
      - ChefSpec
      - Rubocop
      - Foodcritic
    - Chef provisioning
    - Everything else needed to author cookbooks and upload them to the Chef Server

The Chef Workstation

- A computer with ChefDK installed on it is called a Chef Workstation
- Each organization is comprised of:
  - One or more workstations
  - A single server
  - Every node that will be configured and maintained by the chef-client
- Cookbooks and recipes tell the chef-client how each node in your organization will be configured
  - The chef-client does the actual configuration

A Closer Look at the Tools in ChefDK

- The more important tools included are:
  - Berkshelf: a dependency manager for cookbooks
  - chef: a workflow tool for Chef
  - chef-client: the agent that runs Chef
  - chef-vault: tool to encrypt data bag items
  - ChefSpec: a unit testing framework that tests resources locally
  - Fauxhai: a gem for mocking Ohai data in ChefSpec tests
  - Foodcritic: a lint tool for static analysis of recipe code
  - Test Kitchen: an integration testing framework tool that tests cookbooks across platforms
  - knife-spork: a workflow plugin for knife that helps groups of people work together in the same repo
  - Ruby: the reference language for Chef
  - Rubocop ruby style checking tool

Community Tools

- Some of the tools in ChefDK are developed and maintained by members of the Chef community
  - Are considered to be a useful part of the Chef workflow
  - Are maintained outside of the Chef organization

- The community tools that are included with ChefDK are:
  - Berkshelf
  - chef-vault
  - ChefSpec
  - Foodcritic
  - Test Kitchen
  - Rubocop

Setting Up a Chef Workstation

- To provide a consistent environment throughout the course, we recommend taking advantage of one of the provided Linux Academy Server Labs
  - You will be able to follow along with us in most cases as we move around in a remote desktop session or in the terminal
  - You can stop these environments at any time and pick up where you left off later
  - It doesn't cost anything extra to use these Server Labs
    - We'll automatically shutdown the Server Labs when you're not using Linux Academy
    - You can power one or more back up when you need them
  - Learn more by looking at the Server Labs Course Feature on the course page

- Let's go through the exercise of spinning up a Server Lab and installing the ChefDK

Local Cookbook Development

Chef Workstation Setup

Connecting Over SSH

- An alternative to the Web Console is an SSH client
  - Example for macOS/Linux: openssh client
  - Example for Windows: PuTTY

- Use your Server Lab's user account and hostname
  - Example: user@myuser1.mylabserver.com
  - The default port is 22

- You will be presented with a terminal shell environment where you can install applications and run commands

Prerequisite #1: Git

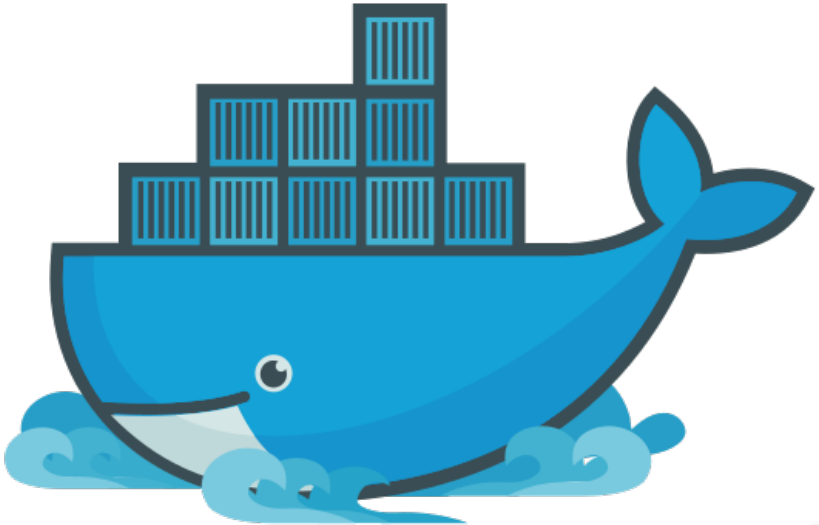
- Git is a distributed version control system
  - There are some commands in ChefDK that depend on git
    - (i.e.: `knife cookbook site install`)

- Git helps teams collaborate. We'll explore this aspect of Local Cookbook Development in an upcoming lesson

- If you want to learn more about git, check out these courses on Linux Academy:
  - Git Quick Start
  - Git and Git lab - Start to Finish

Prerequisite #2: Docker

- Docker is the world's leading software containerization platform
- Docker is great when developers and IT admins want to build, ship and run applications anywhere
- In our context, we'll use Docker for isolated test environments
  - (i.e.: with the Test Kitchen Driver)
- If you want to learn more about Docker, check out these courses on Linux Academy:
  - Docker Quick Start
  - Docker Deep Dive



Local Cookbook Development

Generators

Chef Generate

- Generators are used to quickly generate cookbooks or components of a cookbook. Generators are invoked by using the `chef generate` command.

- What can `chef generate` create?
  - `app` – An application repository
  - `cookbook` – A single single cookbook
  - `recipe` – A single recipe
  - `attribute` – An attributes file
  - `template` – A file template
  - `file` – A cookbook file
  - `lwrp` – A lightweight resource/provider
  - `repo` – A Chef code repository
  - `policyfile` – A policyfile for use with the install/push commands
  - `generator` – A copy of the ChefDK generator cookbook so you can customize it
  - `build-cookbook` – For use with Delivery

Generating with a Generator

- Using `chef generate generator <my_generator>` will create a copy of the ChefDK generator which we can use to customize to our specifications.

- Generating a generator will help enforce standards on the contents of a newly created cookbook or cookbook component to ensure it meets organizational requirements.

- Use cases for generators include:
  - Boilerplate text in README or generated recipes
  - Default licensing choice
  - Commonly used code

Naming Conventions

- When choosing a name for a cookbook be aware of the fact that duplicate cookbook names on Chef Supermarket are not allowed.

- General Advice on naming cookbooks
  - If you plan to share publicly check if your desired name is available on the supermarket first.
  - Use a short organizational prefix for application cookbooks that are part of your organization. If your organization is named GiantCorp, you could use `gc` as a prefix: `gc_postgresql` or `gc_httpd`.
  - Uniqueness in the naming convention can help avoid internal confusion about the purpose and origin of the cookbook.

Local Cookbook Development

Test Driven Development

Why Test?

- Testing ensures that when working through a project either by yourself or with other teams that you don't break your intended behavior.

- Tests can become like documentation for your recipes, acting as a descriptions of the behavior you desire.

- Thorough testing will result in better outcomes.

Different Types of Testing

- Unit testing with ChefSpec requires minimal external dependencies, testing occurs in memory and is therefore very fast. Unit testing is allows you to quickly assert your intent for the behavior the code should exhibit. Effectively you are verifying a resource collection locally in memory with Fauxhai.

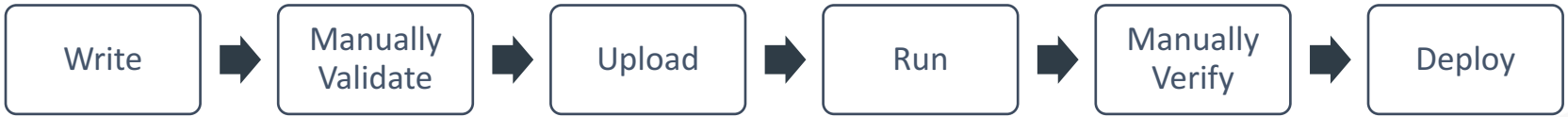
- Functional (Integration) testing is performed via Test Kitchen and Inspec. Integration testing verifies the resulting code by executing a run_list on an actual chef-client and verifying the resulting state of the convergence by examining the system. You can test things you are not able to test with unit testing.

- When both methodologies are used very early in the development process you should have reasonably high confidence in your deployments.

Workflow *Without* Methodical Testing

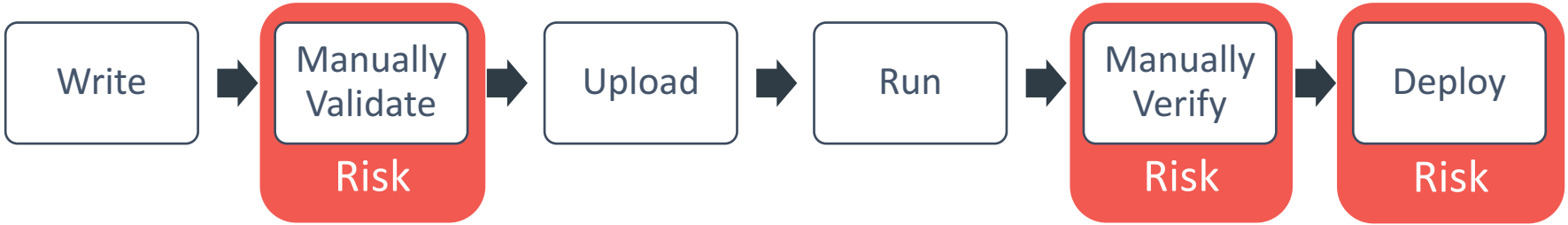
- An example of a workflow which does *not* incorporate a test driven model looks like the following:

1. Write some Chef code.
2. Optionally review the code for correctness and completeness.
3. Upload the code so that it becomes deployable.
4. A chef-client runs the updated cookbook.
5. A manual verification that the system appears to be in the desired state.
6. Promote the cookbook to an environment.



Workflow *Without* Methodical Testing

- By not methodically and carefully testing we are introducing risks into the process because we cannot be sure if we've tested everything.

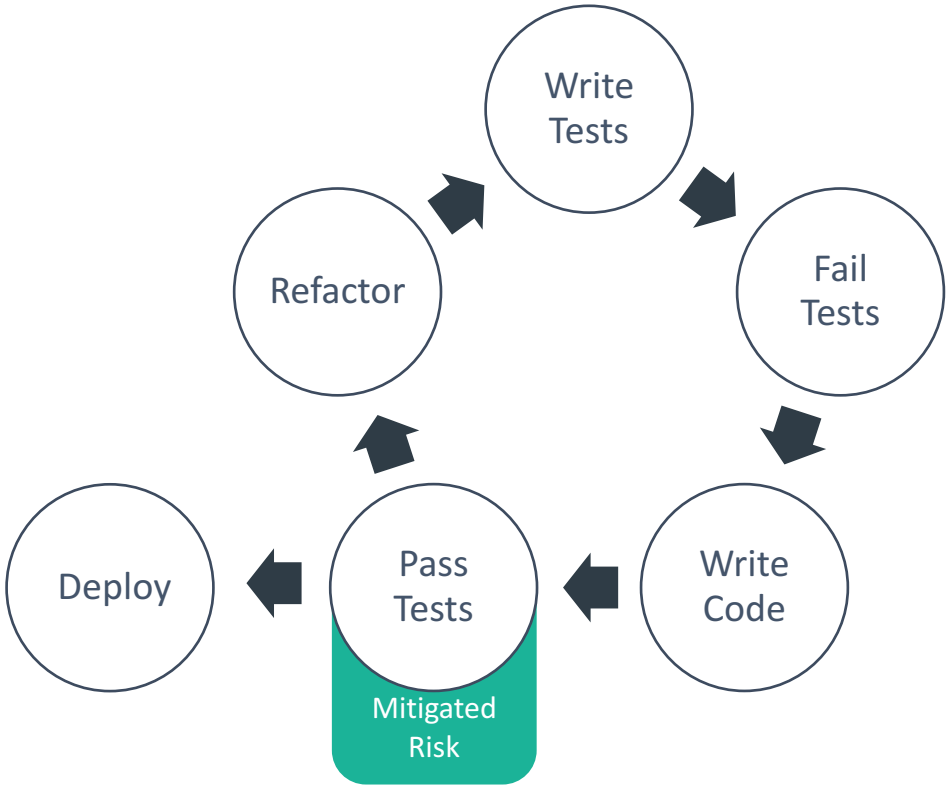


Test Driven Development Workflow

- 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.

1. Define a test for the unit, the smallest testable parts of of our code.
2. Implement the unit by writing the code that performs the operation being tested.
3. Validate that the implementation of the unit satisfies the tests so they are passing.
4. Refactoring as needed to improve.

Test Driven Development Workflow



Summary

- Test Driven Development is a workflow, the ChefDK provides tools to support the workflow.
- Unit testing with ChefSpec provides the fastest means of feedback.
- Integration testing with InSpec is a little slower but runs the tests on an actual instance.

Local Cookbook Development

ChefSpec

About ChefSpec

- ChefSpec is a packaged as part of the Chef development kit and is used for unit testing. ChefSpec provides the ability to quickly test the validity of Chef code with a simulated convergence locally in-memory without running a virtual machine. ChefSpec tests can simulate node attributes and operating systems and can work with search results to provide flexibility in testing scenarios.

- ChefSpec is built on top of RSpec. This means RSpec is the core framework through which tests are executed. Syntax is similar to what we've seen before; you will recognize some of the syntax from InSpec.

- RSpec is a behavior-driven development testing framework. RSpec implements a domain-specific language that reads much like natural language to describe scenarios in which systems are being tested. ChefSpec is built on top of the constructs available in RSpec, so knowing some RSpec is helpful.

ChefSpec Files and Directories

- When invoked via chef generate cookbook the location where the default unit test can be found is relative to the cookbook in a directory called `spec/unit/recipes`. The default recipe will be in a file called `default_spec.rb`. By default, this file will contain a single test which will expect the simulated convergence will not produce an error.

- There is also a file beneath the `spec/unit` directory called `spec_helper.rb`. The spec helper file exists for you to define sets of common expectations or other helpers that would be useful for all your tests. For instance, by default the spec helper will include a reference to the chefspec berkshelf library which will set up and install the necessary dependencies in a temporary directory during simulated convergences.

RSpec and ChefSpec Syntax

- ChefSpec is made available by including a `require 'chefspec'` statement which loads ChefSpec matchers. The default configuration when generating a cookbook is to include this in `spec_helper.rb`.
- The `describe` keyword followed by a cookbook and recipe name tells ChefSpec what to run.
- The `let` keyword is a block which defines how the chef-client run is simulated. SoloRunner or ServerRunner.
- The `it` keyword is the opening block for a test.
- The `expect` keyword sets up an expectation for a resource can be used in conjunction with `to` or `to_not` methods.

```
require 'chefspec'

describe 'example::default' do
  let(:chef_run) { ChefSpec::SoloRunner.converge(described_recipe) }

it 'does something' do
  expect(chef_run).to ACTION_RESOURCE(NAME)
  end
end
```

RSpec and ChefSpec Syntax, Cont.

- The `context` keyword describes a block of tests and can be used to group tests together, this is useful for differentiating sets of tests for specific platforms.

```
describe 'lcd_web::default' do
  context 'CentOS' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '7.2.1511')
      runner.converge(described_recipe)
    end

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

    it 'installs httpd'
      expect(chef_run).to install_package 'httpd'
    end

  end

  context 'Ubuntu' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '16.04')
      runner.converge(described_recipe)
    end

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

    it 'installs apache2'
      expect(chef_run).to install_package 'apache2'
    end

  end
end
```

Executing ChefSpec Tests

- We can execute ChefSpec tests by running them individually or on a case-by-case basis as desired.
- To run all available tests, from the root of a cookbook directory execute the following:

```
chef exec rspec
```

- When running a single test, specify the filename when running the test:

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

- Running with some additional switches can enhance the output for readability:

```
chef exec rspec --color -fd
```

Local Cookbook Development

Test Kitchen Configuration

Test Kitchen Configuration

- Test Kitchen helps automatically test your cookbooks across different platforms and test suites and offers a driver plugin architecture to ensure your code can be tested on as many platforms as you need.

- Configuration for Test Kitchen is defined in a .kitchen.yml file, which is YAML formatted.

- The .kitchen.yml file contains specific syntax to configure the requirements for handling your test cases:
  - Driver
  - Provisioner
  - Verifier
  - Transport
  - Platforms
  - Suites

.kitchen.yml Structure

```
driver:
  name: driver_name

provisioner:
  name: provisioner_name

verifier:
  name: verifier_name

transport:
  name: transport_name

platforms:
  - name: platform-version
    driver:
      name: driver_name
  - name: platform-version

suites:
  - name: suite_name
    run_list:
      - recipe[cookbook_name::recipe_name]
    attributes: { foo: "bar" }
    excludes:
      - platform-version
  - name: suite_name
    driver:
      name: driver_name
    run_list:
      - recipe[cookbook_name::recipe_name]
    attributes: { foo: "bar" }
    includes:
      - platform-version
```

.kitchen.yml Driver

```
driver:
  name: driver_name
```

- Drivers are used to create the platform instances which are used during cookbook testing. There are many driver types available for common virtualization tools and cloud infrastructures.
- Common configurations might use something like Vagrant and VirtualBox to test locally on your workstation, or, if local resources are not available or you have a larger number of tests, you could use a cloud provider like Amazon EC2, or perhaps Docker or one of the many other driver types available.

Vagrant

```
driver:
  name: vagrant
  ...
```

Amazon EC2

```
driver:
  name: ec2
  ...
```

Docker

```
driver:
  name: docker
  ...
```

.kitchen.yml Provisioner

```
provisioner:
  name: provisioner_name
```

- Provisioners specify how the chef-client will be simulated during the testing process.
- The most common configurations are chef-zero and chef-solo.

Chef Zero

```
provisioner:
  name: chef_zero
```

Chef Solo

```
provisioner:
  name: chef_solo
```

- Key functionality of the chef_zero and chef_solo provisioner is the ability to use the require_chef_omnibus setting to install a specific chef version.

```
provisioner:
  name: chef_solo
  require_chef_omnibus: 12.8.1
```

Shell

```
provisioner:
  name: shell
```

.kitchen.yml Verifier

```
verifier:
  name: verifier_name
```

- The verifier configuration defines which test framework will perform tests.

InSpec

```
verifier:
  name: inspec
```

- During this course InSpec will be used to perform integration testing on instances.

.kitchen.yml Transport

```
transport:
  name: transport_name
```

- The transport defines the communication protocol for the execution for remote commands. Among other things, transport settings can include things like username and password, connection retries or port.

SSH

```
transport:
  name: ssh
```

WinRM

```
transport:
  name: winrm
```

.kitchen.yml Platforms

```
platforms:
  name: platform_version
```

- Platforms defines Chef server attributes that are common to the collection of test suites.
- The `platform_version` defines the list of operating system and version on which Kitchen will perform cookbook testing.

Multiple operating systems and versions

```
platforms:
  - name: ubuntu-16.04
  - name: centos-7.2
  - name: centos-6.8
```

.kitchen.yml Suites

```
suites:
  - name: suite_name
    run_list:
      - recipe[cookbook_name::recipe_name]
    attributes: { foo: "bar" }
    excludes:
      - platform-version
  - name: suite_name
    driver:
      name: driver_name
    run_list:
      - recipe[cookbook_name::recipe_name]
    attributes: { foo: "bar" }
    includes:
      - platform-version
```

- Suites is a list of a group of tests which allow the granular combinations of run_lists and platforms to define what is being tested.

Java 6 and Java 7

```
suites:
  - name: java6
    run_list:
      - recipe[java::default]
    attributes: { 'java': { 'jdk_version': '6' } }
    excludes:
      - centos-7.2
  - name: java7
    run_list:
      - recipe[java::default]
    attributes: { 'java': { 'jdk_version': '7' } }
    includes:
      - centos-7.2
```

.kitchen.yml Putting It Together

- Example of a .kitchen.yml

```
driver:
  name: vagrant

provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms
  - name: centos-7.2
  - name: centos-6.8

suites:
  - name: java6
    run_list:
      - recipe[java::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    attributes: { 'java': { 'jdk_version': '6' } }
    excludes:
      - centos-7.2
  - name: java7
    run_list:
      - recipe[java::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    attributes: { 'java': { 'jdk_version': '7' } }
    includes:
      - centos-7.2
```

.kitchen.yml Environments

- What if I need to specify an environment?

- .kitchen.yml using chef_zero

```
...

provisioner:
  name: chef_zero
  environments_path: ../environments

suites:
  - name: prod
    provisioner:
      client_rb:
        environment: production
    run_list:
      - recipe[java::default]
    excludes:
      - centos-7.2
  - name: dev
    provisioner:
      client_rb:
        environment: development
    run_list:
      - recipe[java::default]
    includes:
      - centos-7.2
```

- ../environments/production.json

```
{
 "name": "production",
 "description": "Production environment",
 "cookbook_versions": {
   "my_cookbook": "= 0.1.0"
 },
 "json_class": "Chef::Environment",
 "chef_type": "environment",
 "default_attributes": {
   "java": {
     "jdk_version": ”6"
   }
 }
}
```

.kitchen.yml Environments

- What if I need to specify an environment?

- .kitchen.yml using chef_solo

```
...

provisioner:
  name: chef_solo
  environments_path: ../environments

suites:
  - name: prod
    provisioner:
      solo_rb:
        environment: production
    run_list:
      - recipe[java::default]
    excludes:
      - centos-7.2
  - name: dev
    provisioner:
      solo_rb:
        environment: development
    run_list:
      - recipe[java::default]
    includes:
      - centos-7.2
```

- ../environments/production.json

```
{
 "name": "production",
 "description": "Production environment",
 "cookbook_versions": {
   "my_cookbook": "= 0.1.0"
 },
 "json_class": "Chef::Environment",
 "chef_type": "environment",
 "default_attributes": {
   "java": {
     "jdk_version": ”6"
   }
 }
}
```

.kitchen.yml Putting It Together with Environments

```
driver:
  name: vagrant

provisioner:
  name: chef_zero
  environments_path: ../environments

verifier:
  name: inspec

platforms:
  - name: centos-7.2
  - name: centos-6.8

suites:
  - name: prod
    provisioner:
      client_rb:
        environment: production
    run_list:
      - recipe[java::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    excludes:
      - centos-7.2
  - name: dev
    provisioner:
      client_rb:
        environment: development
    run_list:
      - recipe[java::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    includes:
      - centos-7.2
```

.kitchen.yml with the kitchen-docker Driver

```
driver:
  name: docker
  use_sudo: false
  privileged: true


provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms:
  - name: ubuntu-16.04
    driver_config:
      run_command: /sbin/init
  - name: centos-7
    driver_config:
      run_command: /usr/lib/systemd/systemd

suites:
  - name: default
    run_list:
      - recipe[lcdinit::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    attributes:
```

Local Cookbook Development

Using Test Kitchen

Using Test Kitchen

- Test Kitchen is part of the Test Driven Development model in which tests are written first with the expectation that they will fail. Chef code is then written to make the test instance state match our tests in order to pass. Because of this testing driven model, this means you should use Test Kitchen as early as possible in your Chef development process!

- Test Kitchen is a part of the ChefDK and is invoked with the `kitchen` command.

- Kitchen commands!

Basics of Using Test Kitchen

- Invoking the `kitchen help` command outputs available kitchen commands. Kitchen commands can also be invoked with `kitchen help subcommand`. For example, `kitchen help create` will output help information about the create subcommand within kitchen.
- How can a .kitchen.yml be created?
  - The .kitchen.yml can be created multiple ways:
    - The `kitchen init` command invoked from within a cookbook directory would create a `.kitchen.yml` with a pre-defined run_list containing the name of the cookbook and a configuraiton with its default recipe.
    - The `chef generate cookbook cookbooks/my_cookbook` command would create a `cookbooks/my_cookbook/.kitchen.yml` with a pre-defined run_list containing the my_cookbook::default recipe.
    - The .kitchen.yml can be created via `chef generate app chef_app` and would be located beneath the chef_app/.kitchen.yml
    - You could also create it manually, if desired.
- When run without any arguments, most kitchen commands execute on all related items. You can specify individual instances and use regular expressions for name matching.

kitchen init

- The kitchen init command does the following:
  - Creates a default .kitchen.yml file
    - Includes Vagrant as the default driver
    - Includes chef_solo as the default provisoner
  - Creates the test/integration/default directory
  - Adds .kitchen to chefignore
  - Appends kitchen to the RubyGems file, appends to .gitignore if the directory is under git control, and appends to .thor if necessary.
  - Installs the default Test Kitchen driver if one has not been specified.

kitchen init and Test Directories

- Kitchen test directories
  - When kitchen init is invoked it creates the test/integration/default directory. This directory structure can be used for integration tests.
  - However, when using the Chef command to generate a cookbook, the default location for the inspec tests associated with the default tests are located in test/smoke/default because there are explicit references to this location in .kitchen.yml.
  - You can also create a directory beneath test/integration/<suite_name> and place your tests within that directory and they will be executed without having to be explicitly defined in the .kitchen.yml. Test kitchen knows to look at this location by convention.
  - The location can be explicitly referenced via the .kitchen.yml via specific verifier settings.
  - The `chef generate cookbook` command no longer creates tests in the 'test/recipes/default' directory; it was used in previous versions but has been deprecated, you should be aware of it.

kitchen init with a Specific Driver

- kitchen-vagrant is the default driver, but what if you need to initialize kitchen and include a specific driver?

- The `kitchen help init` command reveals that a driver can be installed with the `kitchen init -D drivername` command if we pass it a valid driver name, or we can use `kitchen init --driver=drivername`

- How to know which drivers are available to install?

  Running the `kitchen driver discover` command will produce a list of available drivers.

- How to initialize your test kitchen configuration with the ec2 driver:

  ```
  kitchen init --driver=kitchen-ec2
  ```

- Drivers can also be installed outside of kitchen by running: `chef exec gem install kitchen-docker`

kitchen list

- The lifecycle of test kitchen instances also have a state which can be viewed by using the `kitchen list` command.

- ”Instance” is a combination of your suite name(s) and platforms being tested. e.g. default-centos-72.
- “Driver” is the configured driver in .kitchen.yml, like Vagrant.
- “Provisioner” is the Chef client implementation. e.g. chef_zero or chef_solo.
- “Verifier” is the testing framework; this course covers Inspec.
- “Transport” is the means by which Test Kitchen communicates with the instances, ssh.

- Last Action can be one of the following depending on what phase of testing the instance is in:
  - Created
  - Not Created
  - Converged
  - Set Up
  - Verified
  - Unknown
- Last Error will record the last error. For instance: `Kitchen::ShellOut::ShellCommandFailed` or `RuntimeError`

Test Kitchen Lifecycle

- The `kitchen` command has a number of sub-commands, which represent a phase of the lifecycle of a test instance.

- The phases of operation are as follows:
  - The first phase is `kitchen create`; this creates and boots your test instance.
  - The second phase is `kitchen converge` which installs a Chef client and executes the run_list in .kitchen.yml
  - The third phase is `kitchen setup` which ensures convergence has occurred successfully and the instances are prepared for testing.
  - The fourth phase is `kitchen verify` which actually runs the tests which have been defined in .kitchen.yml or are implied by convention.
  - When you are finished testing, running `kitchen destroy` removes the test instances.

- All of these operations can be invoked in one single command by using the `kitchen test` command. However, kitchen destroy is implied before `kitchen test` runs.
- The intervening commands are implied if they have not yet been run; for example, converge invokes create if the instance has not yet been created

kitchen create

- The `kitchen create` command will get your instances into a running state. This phase leverages the kitchen driver associated to produce a running test kitchen instance.

- You can invoke the creation of a single instance by specifying that instance name; for example, given a suite name of “prod” to create a centos 6.8 platform version instance, we could use the following command:

```
kitchen create prod-centos-68
```

- The value of the instance name can be obtained with `kitchen list`.

kitchen converge

- The `kitchen converge` command converges the instances in your .kitchen.yml file. When executed, the following occurs:
    - Installs the latest available version of the Chef client using the omnibus installer unless otherwise specified.
    - Uploads cookbooks and required configurations to the instance.
    - Executes a chef-client against the run_list defined in your test suites.
- You can converge multiple times on one or more existing instances without having to reinstall the chef client. This helps when iteratively working on cookbooks.

kitchen setup

- The `kitchen setup` command will ensure that the instance is prepared for tests which, depending on your test framework as in the case of serverspec, would require the installation and preparation of tools on the instance.

kitchen verify

- The `kitchen verify` command will execute tests defined within suites on your instances. This phase invokes the inspec test framework to analyze the state based on the tests you have defined to determine if the conditions on the instance satisfy the tests.

- The verification of a single instance can be run by running `kitchen verify` on the name of the instance.

- The output associated with kitchen verify will let you know how many tests have passed or failed.

kitchen destroy

- The `kitchen destroy` command will remove your test instances by invoking their removal via their associated driver.

- You can invoke the destruction of a single instance by specifying that instance name; for example, given a suite name of “prod”, to create a centos 6.8 platform version instance we could use the following command:

```
kitchen destroy prod-centos-68
```

An Example of an Iterative Development Workflow

1. Set up test kitchen by editing on your kitchen.yml.
2. Write the relevant tests.
3. Run `kitchen verify` (implies create, converge and setup)
4. Examine your failures.
5. Write the Chef code to pass your tests.
6. Run `kitchen converge` to apply your new cookbook code.
7. Run `kitchen verify` to see that you have passed your tests.
8. Iterate on new functionality by writing additional tests.
9. Write Chef to code pass your additional tests.
10. Run `kitchen verify` to ensure your new functionality correctly implemented.
11. When satisfied that your work is complete run `kitchen destroy`.

Test Kitchen Troubleshooting

- Troubleshooting Test Kitchen

- The `kitchen diagnose` command will show you all of the implicit settings for test kitchen's configuration file. This exposes the computed state of .kitchen.yml and will let you examine implicit values for default and computed settings.

- Log in to a test instance with the `kitchen login` command to examine its state. In the case of Vagrant the login session has full access to log in and perform whatever commands are desired.

- Examine `kitchen list`, console output, and any available logs in .kitchen/logs/kitchen.log.

- Use debug logging by running kitchen commands with for example: `kitchen test -l debug`.

Local Cookbook Development

InSpec

About InSpec

- Within Test Kitchen, InSpec functions in .kitchen.yml as a verifier and serves as the test framework for integration testing. Running InSpec tests through Test Kitchen, will allow you to verify your configuration in a test environment.
- InSpec is an open source testing framework with a human-readable syntax that is meant to resemble natural language.

```
describe resource('<name>') do
  it { <expection> }
end
```

- In practice, a simple package verification test would appear as follows:

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

InSpec Test Locations

- By default kitchen-inspec expects tests to be in `test/integration/suite_name` where suite_name is a name identical to a suite listed in .kitchen.yml. The `test/integration` directory is the default value for `test_base_path` in if unset in .kitchen.yml.

- When using the Chef generate commands to generate a cookbook the default location for the InSpec tests is test/smoke/default, and there are explicit references to this location in .kitchen.yml. By automatically including the test/smoke/default reference the default location of test/integration setting is being overidden.

- In prior releases integration tests were located in the ‘test/recipes’, but this is no longer the case.

InSpec Syntax - Resources

- InSpec resources reference different types of resources and their attributes
  - package – Test if a package is installed or not.
  - file – Test file types, their attributes, permissions, file content, checksums, etc.
  - user – Test user existence or properties of a user.
  - group – Test groups and membership.
  - service – Test if a service is installed or running or enabled.
  - port – Test port properties; is a port listening?
  - os – Test platform attributes like operating system family or version.
- These are only a few of common InSpec resource types, there are many more available and can be referenced at inspec.io.

InSpec Syntax - Matchers

- InSpec matchers are used to compare resource values to expectations
  - be – Can be followed by a numerical comparison operator.
  - cmp – Flexible comparison operator.
  - eq – Test for equality between two values of the same type.
  - include – Test if a value is included in a list.
  - match – Check if a string matches a regular expression.

- Each *resource* can define additional resource specific matchers relavant to that *resource*.
  - For example the *package* resource defines a matcher called `be_installed`.
  - The *file* resource defines a matcher called `be_writable`.
  - Check the resource reference documentation for matcher functionality related to that resource.

InSpec Syntax - Expectations

- Expectations come in the form of `should` and `should_not` and are used in conjunction with a matcher that may be resource specific. In practice, that means a simple test to verify a package is installed would look like this:

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

```
Describe user('william') do
  it { should_not exist }
end
```

InSpec User Resource Example

- Common resource example with ‘its’ to compare the value of a setting with a matcher. Additional tests of a resource could look as follows:

```
describe user(‘david') do
  it { should exist }
  its('uid') { should eq 1234 }
  its('gid') { should eq 1234 }
  its('group') { should eq 'root' }
  its('groups') { should eq [’wheel', ’users']}
  its('home') { should eq '/home/david' }
  its('shell') { should eq '/bin/bash' }
end
```

InSpec - Example for file

- Common resource example with ‘its’ to compare the value of a setting with a matcher. Additional tests of a resource could look as follows:

```
describe file(’/var/www/html/index.html') do
  it { should exist }
  it { should be_readable }
  its(‘content’) { should match /hello world/ }
end
```

InSpec – Common Resources Example

- Common resources example

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

describe service('httpd') do
  it { should be_installed }
  it { should be_enabled }
  it { should be_running }
end

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

Local Cookbook Development

Static Code Analysis

Static Code Analysis Tools

- Why perform static analysis?
  - Avoid common mistakes and save time by catching errors quickly.
  - Benefit from more idiomatic code.
  - Best practices for style.
  - Can be integrated into your CI, commit process or even your text editing environment.
  - Customization to support your own specifications.

What is Foodcritic and What Does It Do

- Foodcritic is part of the ChefDK and is invoked with the `foodcritic` command. Foodcritic acts as a linting tool, a correctness checker for Chef code. Foodcritic will check for style, correctness, syntax, best practices, common mistakes and deprecations.

- Foodcritic has built-in rules which are used to check your code. Rules can be run individually or as a group of rules defined by built-in tags. Rules can also be ignored if desired.

- Foodcritic helps catch common cookbook code mistakes that can result in runtime failure. When integrated with a commit hooks or a CI pipeline this prevents errors from making it further than they should.

Using Foodcritic

- Foodcritic tags are rules which have been grouped together and classified based on what they examine. A tag can be checked by using running `foodcritic <path> -t <tagname>` where “tagname” is one of the available tags or individual rule numbers and path is the path to a cookbook or cookbooks. All of the tags and rules are documented at foodcritic.io.

- For instance, running the following: `foodcritic ./ -t correctness` would check all of the available files in the current directory against all rules which have been classified with the “correctness” tag.

- Multiple tags can be run by separating with a comma. For example: `foodcritic ./ -t correctness,style` Will include both the style and correctness tags and all the rules they contain when running food critic.

- Tags can be listed as exceptions. To run all rules except a specific tag prefix the command with a tilde like this: `foodcritic ./ -t ~FC064`

- You can place a .foodcritic file within each cookbook to define the rules that you want to have checked when foodcritic runs. You can also negate rules by prefixing them with a tilde.

Foodcritic Output

- When a problem is detected Foodcritic output will display the rule number, it's description and the file and line number. Below is an example:

  FC064: Ensure issues_url is set in metadata: ./metadata.rb:1

What is Rubocop and What Does It Do

- Rubocop is similar to Foodcritic except that Rubocop evaluates your Ruby code for correctness and style. Out of the box it will enforce many of the guidelines outlined in the Ruby Style Guide. Rubocop does not examine your code for Chef style.

- Rubocop contains the various checks which are referred to as “cops”. Most of the cops in RuboCop are so called style cops that check for stylistic problems in your code. There are also lint cops that check for possible errors and bad practices.

- Rubocop will recurse directories looking for .rb files.

- Rubocop has TODO list functionality that will help you keep track of offenses by creating and masking a list of issues it finds so they can be addressed or ignored.

Rubocop Configuration

- The Rubocop `.rubocop.yml` file might contain settings important to what checks or settings you would like to enforce. These settings will take precedence over defaults that are shipped with Rubocop.
- It is most common to use the same set of rules on all cookbooks for consistency.
- A typical `.rubocop.yml` might appear as follows:

```
AlignParameters:
  Enabled: false

Encoding:
  Enabled: false

LineLength:
  Max: 200

StringLiterals:
  Enabled: false
```

Using Rubocop

- In order to keep track of Rubocop offenses we can generate a `.rubocop_todo.yml` which contains entries that mask all of the offenses it finds. This can be done by executing the `rubocop --auto-gen-config` command.

- The generated `.rubocop_todo.yml` will place the issues it finds into this TODO file and will allow you to toggle them on and off. This file will not automatically be used when running Rubocop without explicitly referencing it from the .rubocop.yml or by using `rubocop --config .rubocop_todo.yml` command.

- In order for this file to automatically be consulted when running the `rubocop` command, you'll need to ensure that a `.rubocop.yml` exists and that the `.rubocop.yml` contains `inherit_from: .rubocop_todo.yml` within.

- The .rubocop_todo.yml will mask the issues that are found so that they can be addressed one by one or ignored.

Rubocop Workflow

- When working with Rubocop you can use a basic workflow to help work through issues it finds.

  1. Within your cookbook run: `rubocop --auto-gen-config`
  2. Add `inherit_from: .rubocop_todo.yml` to `.rubocop.yml` assuming it's in the same directory.
  3. Remove or comment out a single entry within the `.rubocop_todo.yml`.
  4. Run `rubocop` and fix the reported offense.
  5. Verify it has been fixed by running `rubocop` again to ensure it is no longer present.
  6. Repeat steps 3 through 5 until there are no issues left.
  7. Remove the .rubocop_todo.yml when done.

- You might also choose to move offenses which you are not concerned about into the .rubocop.yml to ensure they are not reported in the future.

Rubocop Output

- Rubocop's default output format is a progress meter mode which will display a "." for a clean file and capital letters in the form of C for convention, W for warning, E for error or F for fatal. This shows you at a glance how many issues you have and the kinds of issues.

- Rubocop will display some metrics, including the number of offenses and files checked. Offenses will appear with the filename, line number and character number where the issue is identified. Also included will be the category of error C,W,E or F and a description of the offense in practice this might appear as follows:

```
Inspecting 13 files
..........C.
Offenses:
recipes/default.rb:7:13: C: Space missing after comma.
['net-tools','httpd'].each do |pkg|
            ^
13 files inspected, 1 offense detected
```

Local Cookbook Development

Troubleshooting

Troubleshooting

- A Chef run works in two phases, which ultimately result in establishing your desired state.
    - Compile Phase
    - Execution (Convergence) Phase
- You'll need to have an understanding of:
    - What happens during the compile phase of a Chef client run?
    - What happens during the execution (Converge) phase of a Chef client run?
    - When is pure Ruby code executed?
    - When is Chef code executed?

Chef Run – The Compile Phase

- Each resource in the node object is identified and a resource collection is built. All recipes from the run_list are loaded, and then the actions specified within each of them are identified.

- During the compile phase:
  - All cookbooks are loaded from the run list.
  - Recipes are read and used to build the resource collection.
  - Pure Ruby code is executed
    - Variables get used
    - Arrays get iterated
    - Conditionals get evaluated

Chef Run – The Execution Phase

- The chef-client configures the system based on the order of the resources in the resource collection. Each resource is mapped to a provider which then examines the node and performs the necessary steps to complete the actions.

- During the execution phase:
  - Chef DSL is executed.
  - Any Ruby code inside of a `ruby_block` resource is executed.
  - Guards like `not_if` and `only_if` are evaluated during the execution phase.
  - Property values using `lazy` evaluation are executed.

Chef Run – General Considerations

- Under normal circumstances, resource parameters must be set at compile time, but sometimes a value can't be known until the execution phase. When this is the case, a `lazy` block can be used to have them evaluated during execution time.

- Avoid using pure Ruby code to check for things which haven't yet been executed. Instead use guards like `not_if` and `only_if` which are evaluated during the execution phase.

- There is another exception to standard behavior, the `chef_gem` resource runs its actions immediately, before convergence allowing a gem to be used in a recipe immediately.

Chef Run – Example

- What will happen?

```
file '/tmp/hello.txt' do
  content 'Hello World'
end

if File.exist?('/tmp/hello.txt')
  execute 'the-command'
end
```

- Guards happen in the execution phase where ChefDSL is also executed, thus after the file has been created.

```
file '/tmp/hello.txt' do
  content 'Hello World'
end

execute 'the-command' do
  only_if { File.exist?('/tmp/hello.txt') }
end
```

Chef Run – Notification Considerations

- Remember that by default, notification property timer is `:delayed`, which specifies that a notification should be queued up, and then executed at the very end of the chef-client run.

- Be mindful of the `:before` notification timer which specifies that the action on a notified resource should be run before processing the resource block in which the notification is located.

Chef Run – Notification Considerations

```
lazy_message = 'Hello World'

file ‘lazy_message’ do
  path ‘/tmp/lazy.txt’
  content ”#{lazy_message}”
end

execute 'yum-makecache’ do
  command 'yum makecache’
  notifies :create, 'file[message]', :immediately
  action :nothing
end

package 'bind-utils' do
  action :install
  notifies :run, 'execute[yum-makecache]', :before
end

file 'message' do
  path '/tmp/message.txt’
  content lazy { "#{lazy_message}" }
end

lazy_message = 'Goodbye World’
```

Chef Run – Run Action

- Another method to run something in the compile phase is to use the `.run_action(:some_action)` method at the end of a resource block to run the specified action during the compile phase.
- Action should be set to nothing to prevent execution in the execution phase.

```
execute 'yum-makecache' do
  command 'yum makecache'
  action :nothing
end.run_action(:run)
```

- Resources that are executed during the compile phase cannot notify other resources.

Local Cookbook Development

Cookbook Structure

Cookbook Structure

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- Cookbooks contain a number of files and directories that represent features of that cookbook. Not all of these files and directories are created when generating a cookbook. Certain files and directories are created when using the component specific generator.

- The following commands were used to create the structure to the left and serves only as an example for where files and directories end up being created.

```
chef generate cookbook example_cookbook
chef generate attribute example_attribute
chef generate attribute default
chef generate lwrp example_lwrp
chef generate file example_file
chef generate template example_template
```

Cookbook Components

- **Attributes** – Can be generated with the `chef generate attribute` command.
- **Recipes** – Can be generated with the `chef generate recipe` command.
- **Definitions** – Can be created manually but using custom resources is preferred.
- **Files** – Can be generated with the `chef generate file` command.
- **Libraries** – Can be created manually.
- **Resources** – Can generate custom resources with the `chef generate lwrp` command.
- **Providers** – Can be generated with the `chef generate lwrp` command.
- **Metadata** – Can be generated with `chef generate cookbook`, `repo` or `app` commands.
- **Templates** – Can be generated with the `chef generate template` command.
- **Tests** – Can be generated with recipes when using a generator.
- **Other files** – Some additional files are generated as well including, `Berksfile`, `chefignore`, `.delivery` and `.gitignore`.

Attributes

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- The attributes directory can contain a collection of files which contain attributes or settings which can be used to configure your infrastructure. Attributes function as a way to abstract configuration setting data from the recipe code.
- For each cookbook loaded, the `default.rb` attributes file gets loaded first. Any additional attribute files are loaded in lexical sort order.
- The `default.rb` was generated on purpose and was not created automatically.

Recipes

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- The recipe is the most fundamental configuration element. It contains your configuration instructions.
- Recipes are Ruby code and are a collection of resource names, attribute-value pairs, and actions to which helper code can be added when necessary.

Definitions

- Starting with chef-client 12.5, it is recommended to build custom resources instead of definitions. While it is still possible to use definitions, it is recommended to migrate existing definitions to the new custom resource patterns.

- Definitions would be found within the `definitions` directory within a cookbook.

- Definitions function like a wrapper for resources and are not added to the resource collection. For this reason there are many limitations with definitions. Definitions won't work with common resource properties like `notifies`, `subscribes` and `only_if` or `not_if` guards and they don't support why-run mode.

Files

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- Files are a collection of files which can be added to nodes with the `cookbook_file` resource.
- Files can be arranged in a directory path structure which organizes them for specific host or specific platform targets. Files in the `default` directory would be used on any platform.
- Specificity precedence is as follows:

1. `files/host-$fqdn/$source`
2. `files/$platform-$platform_version/$source`
3. `files/$platform/$source`
4. `files/default/$source`
5. `files/$source`

- A directory structure matching these values would place the file on the desired platform from most to least specific match.

Libraries

- Libraries are located in the `libraries` directory. A library allows arbitrary Ruby code to be included in a cookbook; therefore, anything that can be done with Ruby can be done in a library.

- You can extend existing Chef functionality or create new functionality with a library. Libraries are commonly used to reduce repetition by creating helpers.

- Use cases might include, connecting to a database, interacting with an LDAP provider, creating a custom class or module.

Resources

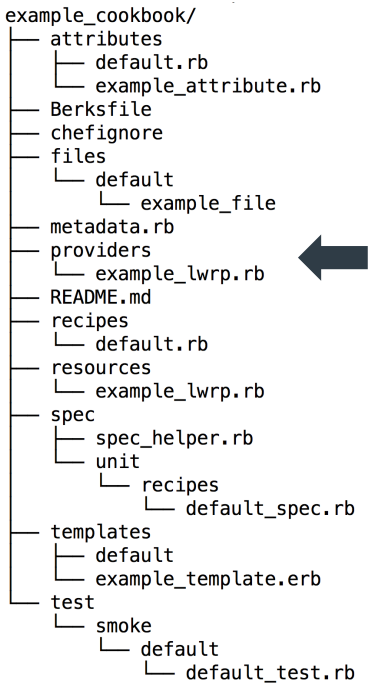
```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- Custom resources are a collection of files located in the `resource` directory which acts as an extension of Chef that allows you to create your own resources.
- Custom resource may be used in a recipe just like the any of the resources that are built into Chef.
- When using custom resources, a cookbook named `website` and a custom resource file named `httpd.rb` is, by default, used in a recipe with `website_httpd`. This behavior can be altered by using `resource_name` method with a value of `:custom_name` which matches what you want to call the resource. This allows the resource to be called using `web_httpd` or whatever arbitrary name you give it.

```
resource_name :web_httpd
```

- You could also create one file in each resource and provider directory called `default.rb` and reference the name as defined.

Providers



- Providers are a collection of files located in the `providers` directory.
- A provider defines the steps to create the desired state and the code necessary to converge the node.
- The naming patterns of lightweight resources and providers are determined by the name of the cookbook and by the name of the files in the `resources/` and `providers/` sub-directories.

Cookbook Structure

Metadata

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- The `metadata.rb` file is created automatically upon cookbook generation and should be located at the top level of every cookbook directory structure.
- The contents of the `metadata.rb` file provide hints to a Chef server to help ensure that cookbooks are deployed correctly.

Templates

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- Templates are located in the `templates` directory and are in Embedded Ruby (ERB) format.
- Templates can be arranged in a directory path structure which organizes them for specific host or specific platform targets. Files in the `default` directory would be used on any platform.
- Specificity precedence is as follows:

1. `templates/host-$fqdn/$source`
2. `templates/$platform-$platform_version/$source`
3. `templates/$platform/$source`
4. `templates/default/$source`
5. `templates/$source`

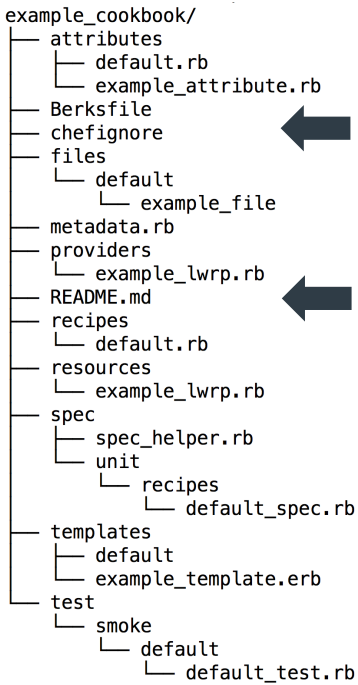
Cookbook Structure

Tests

```
example_cookbook/
├── attributes
│   ├── default.rb
│   └── example_attribute.rb
├── Berksfile
├── chefignore
├── files
│   └── default
│       └── example_file
├── metadata.rb
├── providers
│   └── example_lwrp.rb
├── README.md
├── recipes
│   └── default.rb
├── resources
│   └── example_lwrp.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
├── templates
│   ├── default
│   └── example_template.erb
└── test
    └── smoke
        └── default
            └── default_test.rb
```

- Tests have two separate directories for unit and integration tests and are generated with recipes or during the initial cookbook generation.
- The generated location of InSpec integration tests is within the `test/smoke/default` directory and is referenced within the `.kitchen.yml`. However, the default configuration for kitchen-inspec if no specific location is set is `test/integration/<suite>`. Where suite is the name defined in the .kitchen.yml.
- ChefSpec unit tests are located within the `spec/unit/recipes` directory.

Other Files



- The `chefignore` file is used to prevent specific files being uploaded to the Chef server. It contains file and directory names to ignore when uploading.
- A hidden `.delivery` directory is used for workflow capabilities in Chef Automate and defines how the Chef Automate pipeline will build, test, and deploy a project.
- README.md is the markdown formatted documentation for the Cookbook.
- The `Berksfile` describes the set of sources and dependencies needed to use a cookbook.

Local Cookbook Development

Metadata Anatomy

Metadata Settings

- Metadata settings allow you to define key information about a cookbook. Some settings are common and some are less common.
    - General author cookbook information.
    - Cookbook version numbers.
    - Supported versions of operating systems, Chef client and Ohai.
    - Dependencies for other cookbooks and gems.
    - Recipe descriptions.
    - Source and issues URLs.
    - Less common settings.

Common Settings

- When using a generator, common settings will be created automatically. However, each of these can be changed to suit your organization. All of this information is ideally included in a cookbook so generating it is helpful. Below is an example:

```
name 'org_web'
maintainer 'John Smith'
maintainer_email 'john@example.com'
license 'all_rights'
description 'Installs/Configures lcd_web'
long_description 'Installs/Configures lcd_web'
version '0.1.0'
```

- `name` – The name of the cookbook.
- `maintainer` – Who maintains the cookbook.
- `maintainer_email` – The email address of who maintains the cookbook.
- `license` – The license for distribution.
- `description` – A short description of what the cookbook does.
- `long_description` – A longer more detailed description.
- `version` – The current version of the cookbook.

Version Setting

- The version setting represents the current version of the cookbook in semantic versioning format. The semantic versioning format provides guidelines on when to bump versions and by how much.

- Given a version number MAJOR.MINOR.PATCH, increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards-compatible manner, and
  - PATCH version when you make backwards-compatible bug fixes.

- There are additional version specific metadata settings which will are covered in the versioning lesson.

Supported Platforms and Versions

- There are also version settings which correspond to the supported operating systems.
- The `supports` setting can be configured to declare support for specific platforms and platform versions. To support every version of CentOS, you could declare the following in `metadata.rb`:

```
supports 'centos'
```

- Alternatively you could support anything greater than or equal to CentOS 7:

```
supports 'centos', '>= 7'
```

- Support can be declared multiple times for as many platforms and versions as is necessary:

```
supports 'centos'
supports 'ubuntu'
```

Supported Chef and Ohai Version

- The `chef_version` setting can be configured to declare support for specific Chef client versions; you could declare the following in `metadata.rb`:

```
chef_version '~> 12'
```

- This indicates support for any 12.x version of the Chef client.

- You can similarly configure support for a specific Ohai version:

```
ohai_version "~> 8.22"
```

- This indicates support for the 8.22.x version of Ohai.

Depends

- The `depends` setting requires that a cookbook with a matching name and version exists on the Chef server. The `depends` setting is how the chef-client knows what to download from the Chef server.

- When using this setting you are declaring that your cookbook is dependent upon another cookbook whose name and version match or are evaluated to match what you have specified in `metadata.rb`.

```
depends 'httpd', '~> 0.4'
```

- During the process of development this could be an internal or external dependency, meaning it could be something which is on your Chef server already or something which comes from the Supermarket or something you need to make available via `berks install`.

Gem

- The `gem` setting requires that gems are installed with the chef_gem resource after all the cookbooks have been synchronized but before any loading occurs. This is useful for cases where you have a library which might depend on a gem for some functionality.
- Each gem should be listed once per dependency.

```
gem “poise”
gem “chef-sugar”
gem “chef-provisioning”
```

Recipe

- The `recipe` setting provides a description of the functionality of the recipes within the cookbook.

```
recipe 'lcd_web::users', 'Sets up users'
recipe 'lcd_web::webserver', 'Installs httpd and manages service state'
```

Additional Settings

- If you generated with a generator you will also notice the `issues_url` and `source_url` which are commented out of the metadata.rb file.
- The `issues_url` setting references the URL for the location in which a cookbook's issue tracking is maintained.

```
issues_url 'https://github.com/<insert_org_here>/org_web/issues' if respond_to?(:issues_url)
```

- The `source_url` setting references the URL for the location in which a cookbook's source code is maintained.

```
source_url 'https://github.com/<insert_org_here>/org_web' if respond_to?(:source_url)
```

- These settings integrate with the Supermarket to provide a means for feedback and collaboration for cookbooks.

Less Common Settings

- `privacy` - Specifies that a cookbook is private.
- `provides` - Ordinarily automatically populated, this setting will allow the definition of recipes, definitions or resources.
- You may also encounter the `suggests` and `recommends` options. These do not actually make any changes to the system. They would only provide suggestions or recommendations on additional cookbooks when used. In very recent releases these settings have been removed.

Local Cookbook Development

Versioning

Versioning Cookbooks

- Cookbook versioning is important because the version represents functionality of a cookbook. By enforcing configuration with a specific version you are in control over changes to your infrastructure.

- A cookbook version might be indicative of compatibility or support for third party software, correcting a bug or adding a feature.

- A cookbook version always takes the form x.y.z where x, y and z are decimal numbers. Alphanumeric combinations are not allowed. Version numbers with more than 3 digits are also not allowed.

- Approaches to help with cookbook version change and the process of versioning.

Semantic Versioning

- Semantic versioning format provides guidelines on when to change versions and what digit to adjust, depending on the impact of the change.

- Given a version number MAJOR.MINOR.PATCH, increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards-compatible manner, and
  - PATCH version when you make backwards-compatible bug fixes.

- An example of this might be `1.1.0`, which might be incompatible or have major functionality changes when compared with version `2.0.0`.

Version Constraints

- Version constraints allow the ability to obtain or avoid a specific version or range of versions. These constraints are applied to a metadata function which describes the desired functionality.
- The syntax applies an operator to scope the version to obtain the desired effect. These are common operators like equal to, greater than, less than but there are some additional optimistic and pessimistic operators.

=	equal to
>	greater than
<	less than
>=	greater than or equal to; also known as “optimistically greater than”, or “optimistic”
<=	less than or equal to
~>	approximately greater than; also known as “pessimistically greater than”, or “pessimistic”

Version Constraints

- If no constraint is specified, the default constraint used is `>= 0.0.0` which means the version used will be versions greater than or equal to the specified version. That means version `1.0.0` matches, as does `2.0.0` or `2.5.0` or `5.2.1`. The highest available version will match and will be used.

- The pessimistic version constraint uses a tilde combined with one of the standard operators and appears as follows: `~> 2.5.0`. This will match versions greater than `2.5.0`, for example `2.5.1`, `2.5.2` etc, but not `2.6.0`. This same functionality applies when using only two digits of significance in the version number.

- Another pessimistic constraint would match values less than a specific version. For instance, `< 2.3.4` matches anything available below the target number.

Metadata Functions

- There are functions which can inhibit or influence the behavior when used within `metadata.rb`.

- `conflicts` – Declares a cookbook is in conflict.
- `depends` – Declares a dependency on a cookbook.
- `provides` – A list of recipes or definitions which would not be covered via auto-generation.
- `recommends` – Adds a dependency recommendation
- `replaces` – Declares that this cookbook replaces another.
- `suggests` – Adds a dependency on another cookbook as a suggestion.
- `supports` – Declares a supported platform.

Metadata - Conflicts

- The conflicts functionality will declare that the cookbook you're working with has a conflict with an existing cookbook or cookbook version. Any cookbooks in conflict which match the version constraint are not sent to the node by the Chef server during a chef-client run.

```
conflicts 'mysql', '> 7.1.0'

conflicts 'mysql'
```

Metadata - Depends

- The `depends` functionality will declare the cookbook you require for your cookbook to successfully converge. It is very important that this field is accurate. The dependency below will require that the mysql cookbook is available.

```
depends ‘mysql’, ‘> 7.1.0’
```

```
depends ’mysql’
```

Metadata - Provides

- The `provides` functionality will allow you to add a recipe, definition, or resource that is provided by a cookbook. This list is ordinarily automatically populated, so generally it's not required.

- In the case of recipes:

```
provides 'myapp::auth'
provides 'myapp::web'
```

- In the case of a resource:

```
provides 'service[myappsvc]'
```

Metadata - Recommends

- The `recommends` functionality will provide a recommendation for a cookbook which is not actually required. The cookbook will still work if the recommendations are not available.

```
recommends 'mysql', '> 7.1.0'
```

Metadata - Replaces

- The `replaces` functionality will indicate this cookbook replaces an existing cookbook, which indicates it can be used in place of that cookbook.

```
replaces 'mysql'

replaces 'mysql', '> 7.1.0'
```

Metadata - Suggests

- The `suggests` functionality will indicate this cookbook suggests, but does not require, another cookbook. It is documented as being *"weaker"* than recommends. But remember that neither of them will cause the cookbook to fail if the defined cookbook is not available.

```
suggests 'mysql', '> 7.1.0'
```

Metadata - Supports

- The `supports` functionality will declare support for a platform or platform version.

```
supports ‘centos’
```

- Alternatively you could support anything greater than or equal to CentOS 7:

```
supports ‘centos’, ’>= 7’
```

- Support can be declared multiple times for as many platforms and versions as is necessary:

```
%w{ debian ubuntu redhat centos fedora amazon }.each do |os|
  supports os
end
```

Managing and Controlling Changes to Cookbooks

- To prevent unwanted or accidental changes to a cookbook, the cookbook version can be frozen when uploaded to the Chef server, which prevents updates from occurring on that cookbook version. The example below prevents changes to the cookbook version uploaded.

```
knife cookbook upload myco_app --freeze
```

- Other versions can be still be uploaded, but none with this specific version unless forced:

```
knife cookbook upload myco_app --force
```

- Using a freezing process for cookbooks associated with an environment can help avoid accidental overwrites.

Managing and Controlling Changes to Cookbooks

- The Knife Spork tool was developed by Etsy and has been included with the ChefDK that help provide a workflow for multi-developer Chef environments. The `knife spork` command allows you to more quickly review and manipulate cookbook version information.

- Knife Spork can be configured with a `.spork-config.yml` to allow the configuration of environment specific settings
- The `knife spork check` command can help identify local and remote versions to determine if there are unfrozen remote versions which might clash with your version.
- The `knife spork bump` command can be used to version a cookbook without updating the `metadata.rb` file.
- The `knife spork upload` command automatically makes a cookbook frozen upon upload.
- The `knife spork promote` command will set a version constraint on an environment for a cookbook.

Local Cookbook Development

Attributes

What are Attributes and Where are They Defined?

- Attributes are the details about a node. They are used by the chef-client to understand node state over time. The chef-client must know the current state of the node, the previous state of the node after the last run, and what the node state should become at the end of the current run.
- Candidates for attributes are abstractions of configuration information for an application, values for tunable application parameters.

- Attributes are obtained by the chef-client in the following locations:
  - Node information from Ohai
  - Attribute files from within cookbooks
  - Recipes also within cookbooks
  - Environments
  - Roles

Attribute Behavior on Chef Run

- The chef-client will obtain the node object from the Chef server, which has all the attribute data from the last chef-client run. After that is complete, all attributes except `normal` are reset. This implies a rebuild process in which attributes from their various sources are consolidated and evaluated for precedence.

- Attributes support multiple levels of precedence. An attribute may have multiple values internally, but will only return the highest precedence value when read.

- If an attribute is uniquely identified it is consolidated into the node object; if this is a single instance there would be no precedence to evaluate.

- At the conclusion of the next chef-client run the node object is sent back to the Chef server and it becomes indexed for search and is stored until the next chef-client run.

Attribute Types

- There are six attribute types which can be used to determine a value that is applied during a chef-client run. These represent the precedence of an attribute: `node.<type>['attribute_name'] = 'value'`

`default`	Default has the lowest precedence and should be used as often as possible.
`force_default`	When defined in a cookbook, takes precedence over a default value in a role or environment.
`normal`	The `normal` attribute takes precedence over `default`, it persists and isn't reset.
`override`	Override has a higher precedence than `default`, `force_default` and `normal` attributes.
`force_override`	Takes precedence over an `override` attribute set by a role or an environment.
`automatic`	The `automatic` attribute is identified by Ohai

Automatic Node Attributes

- Automatic node attribute types that come from Ohai include vital node specific information, the most common of which would be discoverable vital information about the system on which the chef-client has run. The attributes collected by Ohai are unmodifiable by the chef-client directly.

```
node['platform']
node['platform_family']
node['platform_version']
node['ipaddress']
node['macaddress']
node['fqdn']
node['hostname']
node['domain']
node['recipes']
node['roles']
```

- A simple use case for these kinds of attributes is for conditional platform detection:

```
if node['platform'] == 'ubuntu'
  # ubuntu specific work
end
```

Attribute Definition and Reference

- When you see an attribute defined in an attributes file there is an implied reference to `node` which precedes the the precedence type of `default`.

  This attribute:

  ```
  default['java']['jdk_version'] = '6'
  ```

  Represents the same thing as:

  ```
  node.default['java']['jdk_version'] = '6'
  ```

- When referencing an attribute you can use `node['java']['jdk_version']` to obtain the merged attribute information.
- When defining within a recipe use node.default['attribute_name'] = 'value'

Attribute Precedence

1. An automatic attribute identified by Ohai at the start of the chef-client run.
2. A force_override attribute located in a recipe.
3. A force_override attribute located in a cookbook attribute file.
4. An override attribute located in an environment.
5. An override attribute located in a role.
6. An override attribute located in a recipe.
7. An override attribute located in a cookbook attribute file.
8. A normal attribute located in a recipe.
9. A normal attribute located in a cookbook attribute file.
10. A force_default attribute located in a recipe.
11. A force_default attribute located in a cookbook attribute file.
12. A default attribute located in a role.
13. A default attribute located in an environment.
14. A default attribute located in a recipe.
15. A default attribute located in a cookbook attribute file.

Attribute Precedence

	Attribute Files	Node / Recipe	Environment	Role
`default`	1	2	3	4
`force_default`	5	6		
`normal`	7	8		
`override`	9	10	12	11
`force_override`	13	14		
`automatic`	15			

Attribute Examples

- In attributes/default.rb

```
default['app']['language'] = 'perl'
```

- In recipes/default.rb which overrides the attributes file.

```
node.default['app']['language'] = 'ruby'
```

Attribute Examples

- In environments/development.json which overrides the recipe.

```
{
"name": "development",
"description": "Developemnt Environment",
"chef_type": "environment",
"json_class": "Chef::Environment",
"default_attributes": { "app": { "language": "php-cli" } },
"override_attributes": {},
"cookbook_versions": {
  "lcd_web": "= 0.1.0"
}
}
```

Local Cookbook Development

Common Resources

Common Resources

- You should be familiar with some of the common resources and their functionality.
  - `package`
  - `service`
  - `directory`
  - `user` and `group`
- Multiple file related resources:
  - `file`
  - `cookbook_file`
  - `remote_file`
  - `template`
- Execution resources:
  - `execute`
  - `script, bash, ruby, python etc.`
  - `shell_out`

Common Properties

- There are properties which are a part of the common functionality available to all resources.

  - `ignore_failure` - Can be set true or false, continue if the resource fails for any reason.
  - `provider` - Provider allows for the override of the auto-detected provider for a resource.
  - `retries` - The number of times to catch exceptions and retry.
  - `retry_delay` – The delay in seconds for retry.
  - `sensitive` – When set to true, will ensure that sensitive resource data is not logged by the chef-client.
  - `supports` – Declares some functionality as hints to a provider about additional capabilities.

```
service 'app_service' do
  provider Chef::Provider::Service::Upstart
  supports :status => true, :restart => true, :reload => true
  action [ :enable, :start ]
end
```

Package

- The package resource will leverage the package provider associated with the running platform to install a package.

```
package node['database']['name'] do
  action :install
end

package 'Install Apache' do
  case node['platform']
  when 'redhat', 'centos'
    package_name 'httpd'
  when 'ubuntu', 'debian'
    package_name 'apache2'
  end
end
```

Service

- The service resource will leverage the provider associated with the running platform to manage a service.

```
service 'http_service' do
  case node['platform']
  when 'redhat', 'centos'
    service_name 'httpd'
  when 'ubuntu', 'debian'
    service_name 'apache2'
  end
  action [:enable, :start]
end
```

Directory

- The directory resource will manage directories. Keep in mind that when using the "recurse" parameter, only the leaf directory, the last one in the tree, will have permissions and ownership set according to the recipe code.

```
app_path = "/opt/app/conf/#{node['fqdn']}"

directory app_path do
  user node['app']['user']
  group node['app']['group']
  recurse true
end
```

- To ensure the path has the same owner and group or permissions:

```
['/opt/app', '/opt/app/conf', "/opt/app/conf/#{node['fqdn']}" ].each do |path|
  directory path do
    user node['app']['user']
    group node['app']['group']
  end
end
```

User

- The user resource manages users. The associated actions for user are `create`, `lock`, `manage`, `modify`, `nothing`, `remove` and `unlock`. The default action is `create`.
- Like with any resource, you may often see these abstracted with configuration values derived from attributes.

```
user node['app']['user'] do
  group node['app']['group']
end
```

- As well as with the individual parameters you would expect to find for a user:

```
user 'jsmith' do
  manage_home true
  comment 'John Smith'
  uid '1001'
  gid '1001'
  home '/home/jsmith'
  shell '/bin/bash'
  password ' $1$zP32a/HN$W2XYcPyPG839EJEUEwSHY1'
end
```

Group

- The group resource manages groups. The associated actions for user are `create`, `manage`, `modify`, `nothing`, and `remove`. The default action is `create`.
- Like with any resource, you may often see these abstracted with configuration values derived from attributes.

```
group node['app']['group']
```

- As well as with the individual parameters you would expect to find for a group:

```
group 'www-data'
  do action :modify
  members 'jsmith'
  append true
end
```

- When using the `append true` parameter the value of members is additive and is appended to the list of existing members. When append is set to true, any members which are also defined in `excluded_members` will be removed from the list. The default value of `append` is false, which means that it is reset exactly to the list of members.

File Related Resources

- File related resources have a set of common actions and most of their properties are identical since you need to be able to perform common activities like setting permissions, ownership, path information etc.
  - Use the `file` resource to manage files on the node.
  - Use the `cookbook_file` resource to copy a file from a cookbook's /files directory to the node.
  - Use the `template` resource to create a file based on a template located in the cookbook /templates directory.
  - Use the `remote_file` resource to transfer a file to a node from a remote location.

- Actions
  - Default action is to `:create` the file.
  - The `:create_if_missing` action will only create if the file does not exist.
  - File removal is invoked with `:delete`
  - The `:nothing` action will await notification.
  - The `:touch` action will update the atime and mtime for the file.

File Related Resources – file

- The `file` resource:

```
file '/tmp/message' do
  owner 'root'
  group 'root'
  mode '0755'
  action :create
end
```

Attributes can be referenced to supply values to

```
file '/tmp/appfile' do
  action :create_if_missing
  owner node['app']['user']
  mode '0400'
end
```

File Related Resources – cookbook_file

- The `cookbook_file` resource can be used to transfer files from the files subdirectory within a cookbook to the specified path on the server running the chef-client. All of the same functionality of file, except you're able to copy a file from the cookbook to the client. A checksum is computed for local file in the cookbook and the remote file. If the remote file does not match, the file will be uploaded.

```
cookbook_file '/var/www/html/index.php' do
  source 'index.php'
  owner 'apache'
  group 'apache'
  mode '0644'
  action :create
end
```

File Related Resources – remote_file

- The `remote_file` resource can be used to transfer files from a remote location. There are a number of protocols available which may influence the parameters required when defining the resource. These could be file, ftp, sftp, a local file path on the chef-client.

```
remote_file '/var/www/html/index.html' do
  source 'http://www.example.com/index.html'
  owner 'apache'
  group 'apache'
  mode '0644'
  action :create
end
```

- There are a number of additional additional properties for this resource, for example: The use of the `checksum` property will check if the checksum of the remote file matches the chef-client's own checksum and if they match it will not be downloaded.

File Related Resources – template

- The `template` resource allows for the creation of a file based on an Embedded Ruby (ERB) template file that dynamically generates static text files and places them on host where the chef-client has run. The functionality is very similar to that of the `cookbook_file` resource except the source file location and formatting.

```
template'/var/www/html/index.html' do
  source 'index.html.erb'
  owner 'apache'
  group 'apache'
  mode '0644'
  variables({
    :greeting_scope => 'World'
  })
end
```

- Where attribute `node.default['greeting'] = 'Hello'` is set and `templates/index.html.erb` contains:

```
<%= node['greeting'] %> <%= @greeting_scope %> from <%= node['fqdn'] %>
```

Execute

- The execute resource allows for the execution of a single command. Execute is generally not idempotent resource but when used with `only_if` and `not_if` it can be guarded to ensure idempotence.

```
execute 'systemctl restart httpd' do
  not_if 'curl -s http://localhost | grep "Hello World"'
end


execute 'app-installer'
  cwd '/opt/app/install'
  command './installer --install'
  user 'appuser'
  only_if { File.exist?('/opt/app/install/prereqs-complete.log') }
end
```

- The execute resource also supports the creates property which can be used in place of file detection guards.

```
  only_if { File.exist?('/opt/app/install/prereqs-complete.log') }

Could be replaced with

  creates '/opt/app/install/prereqs-complete.log'
```

Script Interpreters

- For scenarios in which you need to execute and entire multi-line script you can use common interpreter resources by referencing either the `script` resource or the interpreter name as the resource.
- The script resource allows the interpreter parameter to be set, whereas with the resources named for their interpreter it is implied.

```
script 'extract_module' do
  interpreter 'bash'
  environment 'PATH' => "/my/path/to/bin:#{ENV['PATH']}"
  code <<-EOH mkdir -p /opt/app/install
    tar xzf /tmp/installer.tgz -C /opt/app/install
    /opt/app/install/run.sh
    EOH
  not_if { ::File.exist?(/opt/app/install/install-complete.log) }
end
```

ShellOut and shell_out

- The `Mixlib::ShellOut` class provides a simplified interface to shelling out and collecting both standard out and standard error and providing full control over environment, working directory, uid, gid, etc. It will accept commands or command fragments.

```
find = Mixlib::ShellOut.new("find . -name '*.rb'")
find.run_command
puts find.stdout
```

- The `shell_out` method runs a command on the system and will raise an error if the command fails and output to the tty when the log level is debug.

```
modules = shell_out("apachectl -t -D DUMP_MODULES")
puts modules.stdout
```

Local Cookbook Development

Templates

About Templates

- Two things need to be present for templates to function:
  - A `template` resource declared in a recipe.
  - A template file associated with the declared resource.

- Templates are in Embedded Ruby (ERB) format and contain both Ruby expressions and statements in addition to the content required to model a file.

- The template file can be in any of the following directories beneath `templates`. The first pattern that matches is used to identify the template which should be used. This is referred to as file specificity.

```
1. /host-$fqdn/$source
2. /$platform-$platform_version/$source
3. /$platform/$source
4. /default/$source
5. /$source
```

Generating a Template

- Generate a template by using a generator. Alternatively, the template can be created manually and placed in the desired location beneath the templates directory for file specificity.

```
chef generate template cookbooks/lcd_web index.html
```

- This will have generated a file called `templates/index.html.erb`. This represents the least specific match in terms of specificity.

Template Resource

- The `template` resource defines the destination of the file, the source template name and properties of that file like ownership and permissions. The template resource supports common file actions like, `create`, `create_if_missing`, `delete`, `nothing` and `touch`.

```
template'/var/www/html/index.html' do
  source 'index.html.erb'
  owner 'apache'
  group 'apache'
  mode '0644'
  variables({
    :greeting_scope => 'World'
  })
end
```

- Where attribute `node.default['greeting'] = 'Hello'` is set and `templates/index.html.erb` contains:

```
<%= node['greeting'] %> <%= @greeting_scope %> from <%= node['fqdn'] %>
```

Template Syntax

- When the ability to print the value of a variable or Ruby expression is required, use the `<%=` and `%>` tags within the template. The "=" indicates that the resulting evaluation should be output.

```
<%= @template_variable %>
```

- When outputing, `<%= -%>` can be used to trim line breaks. Note the '-'.

- To use Ruby conditional logic, use `<%` and `%>` which are non-printing tags that will execute your logic.

```
<% if @webservers.each do |server| %>
  <%= server %>
<% end %>
```

```
<% if @greeting == "Hello" %>
  <%= @greeting %> <%= @greeting_scope %> from <%= @fqdn %>
<% end %>
```

Partial Templates

- Templates can also be independent files which are can be combined together using partial template files and specifying that they be rendered within the template file. These are invoked inside the template with the render keyword.
- Given a resource like the following:

```
template '/var/www/html/index.html' do
 source 'index.html.erb'
 owner 'apache'
 group 'apache'
 variables(
   greeting_scope: 'World',
   greeting: node['greeting'],
   fqdn: node['fqdn']
 )
end
```

- And a template file which contains the following:

```
<%= render "header.html.erb" %>
<% if @greeting == "Hello" %>
<%= @greeting %> <%= @greeting_scope %> from <%= @fqdn %>
<% end %>
<%= render "footer.html.erb" %>
```

Partial Templates Continued

- The `templates/header.html.erb` contains the following:

```
<html>
<head>
<title><%= node['fqdn'] %></title>
<body>
```

- The `templates/footer.html.erb` contains the following:

```
<p>Goodbye World!</p>
</body>
</html>
```

Partial Templates Continued

- The resulting output will combine all the template content in the order of its definition.

```
<html>
<head>
<title>cc9268dc6734</title>
</head>
<body>

Hello World from cc9268dc6734

<p>Goodbye World!</p>
</body>
</html>
```

Local Cookbook Development

Libraries

About Libraries

- Library code exists within the `/libraries` directory as a collection of files.
- The primary use case is for reusable functionality in pure Ruby code. You could, for example, connect to a database and make an LDAP query or anything else possible within Ruby.
- You can also include simple helper code for commonly used functionality.

```
def index_exists?
  ::File.exists?("/var/www/html/index.html")
end


 execute 'systemctl start httpd' do
  only_if { index_exists? }
end
```

- By default you can't use core Chef DSL resources in libraries.

About Libraries

- Helpers can come in many forms to make repetitive tasks easier. You could have a set of helper functions in `libraries/helpers.rb`. We can create a namespace which can be included and referenced elsewhere when necessary.

```
module LcdWebCookbook
 module Helpers
   def platform_package_httpd
     case node['platform']
     when 'centos' then 'httpd'
     when 'ubuntu' then 'apache2'
     end
   end

   def platform_service_httpd
     case node['platform']
     when 'centos' then 'httpd'
     when 'ubuntu' then 'apache2'
     end
   end
 end
end
```

About Libraries

- The module definitions are arbitrary but represent a way to contain within some namespace.

```
module LcdWebCookbook
 module Helpers
 ...
```

- This could be included in a resource or recipe code by including them via the file within libraries or in a resource file or recipe file.

```
Chef::Recipe.include(LcdWebCookbook::Helpers)
Chef::Resource.include(LcdWebCookbook::Helpers)
```

Local Cookbook Development

Custom Resources

About Custom Resources

- Custom Resources offer the ability to extend Chef by creating your own custom reusable components in Chef. They can be used to encapsulate an entire group of resources into one simpler-named resource with custom parameters and input validation.

- The primary benefit is reusable functionality using a simpler DSL-based syntax. Only a few key settings are needed to define a custom resource. In a basic form, a name, one or more properties and an action to accomplish the desired state is all that is required.

- Custom resource names are automatically derived unless the `resource_name` property is set. Otherwise it is, by default, the name of the cookbook followed by the name of the resource file with an underscore in between. For example, if you had a cookbook called "apache" and a resource called "vhost" the default name for the custom resource would be `apache_vhost`.

- Creating a dependency would be required when consuming a custom resource in another cookbook.

About Custom Resources

```
resource_name :hello_httpd
property :greeting

action :create do
  package 'httpd' do
    action :install
  end

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

  template '/var/www/html/index.html' do
    source 'index.html.erb'
    owner 'apache'
    group 'apache'
    variables(
      greeting_scope: node['greeting_scope'],
      greeting: greeting,
      fqdn: node['fqdn']
    )
  end
end
```

Using Custom Resources in Other Cookbooks

- To use this custom resource in another cookbook, a dependency must be configured in `metadata.rb`. Assuming my cookbook name is lcd_web, this would appear as follows in the metadata file of another cookbook:

```
depends 'lcd_web'
```

- Additionally, I would need to configure the `Berksfile` with a reference to the location of the `lcd_web` cookbook.

```
cookbook 'lcd_web', path: '~/chef/cookbooks/lcd_web'
```

- I also have another problem, because the resource uses a template and also uses partial templates. In order to make the resource works properly, I need to include where the template should be sourced from because, by default, they are assumed to be located within the local cookbook.

About Custom Resources

```
resource_name :hello_httpd
property :greeting, :kind_of => String

action :create do
  package 'httpd' do
    action :install
  end

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

  template '/var/www/html/index.html' do
    cookbook 'lcd_web'
    source 'index.html.erb'
    owner 'apache'
    group 'apache'
    variables(
      greeting_scope: node['greeting_scope'],
      greeting: greeting,
      fqdn: node['fqdn']
    )
  end
end
```

About Custom Resources

- The template content for partials must also be adjusted:

```
<%= render "header.html.erb", cookbook: 'lcd_web' %>
<% if @greeting == "Hello" %>
<%= @greeting %> <%= @greeting_scope %> from <%= @fqdn %>
<% end %>
<%= render "footer.html.erb", cookbook: 'lcd_web' %>
```

- An attribute value should be also be set to ensure `greeting_scope` is set in `attributes/default.rb`, per the code.

```
default['greeting_scope'] = 'World'
```

- The recipe in the new cookbook can consist of only the custom resource name:

```
hello_httpd 'greet world' do
  greeting "Hello"
  action :create
end
```

Local Cookbook Development

Cookbook Disposition

Cookbook Disposition

- When using a generator you have the ability to generate repo, app and individual cookbooks, which take slightly different approaches to the management and organization of cookbooks. You are also free to create elements of a cookbook yourself without using a generator.

- The ChefDK tools support multiple workflows and ways of arranging, maintaining and testing cookbook code. How your cookbooks and repositories are structured or generated may influence maintainability, deployment and testing.

- A cookbook might represent a single piece of software but be collocated with many other cookbooks in the same Chef repository. There may be cookbooks that have little to do with each other but are within the same source control repository. You may also have external dependencies that need to be managed in addition to the constraints described already.

Monolithic Repository

- All Chef-related content is tracked in one source control repository, including any dependencies. Dependencies are branches which can be tracked for upstream modifications.

```
chef generate repo my_repo
chef generate my_repo/cookbooks/my_cookbook

knife cookbook site download httpd
knife cookbook site install httpd
...
```

- This methodology has the advantage of self-containment, where the application and infrastructure code are in the same place.

- This is challenging longer term, when multiple people need to work on a large project, incorporating changes and testing the cookbook components. Sharing some functionality of a single cookbook might be more difficult.

One Repository Per Cookbook

- Another model asserts that all Chef cookbooks are independent from one another and that they can be built and maintained in isolation from one another.
- This model assumes that dependencies can be resolved as required and don't need to be stored directly with the cookbook. Upstream changes can be incorporated with versioning providing controls over when major changes might dictate a new release.

```
chef generate cookbook my_cookbook
```

- This methodology has the advantage of a smaller footprint and easier incorporation of upstream dependency changes. Sharing and collaboration is easier and changes to cookbooks which represent a dependency can be done independently. Automated testing may be easier.
- Testing is done in isolation, additional unit tests may be required.

Local Cookbook Development

Wrapper Cookbooks

What is a Wrapper Cookbook?

- A wrapper cookbook is a cookbook whose functionality is encapsulated so that its settings can be adjusted for your environment.
- Wrapper cookbooks allow the source cookbook to remain unmodified. You can configure attributes within the wrapper cookbook which reflect the configuration settings or tunables of the application. These attributes are specific to your environment, and if they are identical to those shipped with the source cookbook they will override those settings.
- When using wrapper cookbooks it's a generally accepted practice to prefix your cookbook name with an organizational identifier of some significance so that the wrapper cookbook name is something like `company_cookbook`.

```
chef generate cookbook lcd_haproxy
```

Wrapper Cookbook Setup

- Wrapper cookbooks require a dependency on the cookbook you're attempting to create a wrapper for. This means that in the case of a wrapper for haproxy you'd edit `lcd_haproxy/metadata.rb` to include a dependency.

```
depends 'haproxy', '= 3.0.2'
```

- This means that the `haproxy` cookbook will also be delivered to the chef-client in addition the wrapper cookbook.

- Our next steps will be to ensure that our own settings are included to override the default values provided by the cookbook.

Wrapper Cookbook Attributes

- The supermarket page for haproxy documents its usage. It's always good to consult this documentation so that you know what attributes are available and how you use the cookbook recipes. Some cookbooks will automatically do things like installation upon including the default recipe, others avoid using the default recipe altogether.
- By creating some default attributes in `attributes/default.rb`, according to the documentation, this will override the defaults and create a configuration file with the load balancer members.

```
default['haproxy']['members'] = [{
  'hostname' => 'webserver1',
  'ipaddress' => '10.1.1.3',
  'port' => 80,
  'ssl_port' => 443,
}, {
  'hostname' => 'webserver2',
  'ipaddress' => '10.1.1.4',
  'port' => 80,
  'ssl_port' => 443,
}]
```

Wrapper Cookbook Recipe

- The default recipe of the lcd_haproxy cookbook would contain the following:

```
include_recipe 'haproxy::manual'
```

- Technically, this is all that is needed to install and configure and deploy haproxy.

Wrapper Cookbook Testing

- The suites portion of the test kitchen configuration could be configured with multiple webserver instances which have different run list than the `loadbalancer` suite.

```
suites:
  - name: loadbalancer
    driver:
      run_options: --network=testnet --ip=10.1.1.2
      forward: 80:80
    run_list:
      - recipe[lcd_haproxy::default]
    verifier:
      inspec_tests:
        - test/smoke/default
    attributes:
  - name: webserver1
    driver:
      run_options: --network=testnet --ip=10.1.1.3
    run_list:
      - recipe[lcd_web::default]
    verifier:
      inspec_tests:
        - test/smoke/default
  - name: webserver2
    driver:
      run_options: --network=testnet --ip=10.1.1.4
    run_list:
      - recipe[lcd_web::default]
    verifier:
      inspec_tests:
        - test/smoke/default
```

Wrapper Cookbook Testing

- Since the lcd_web cookbook is outside of the test kitchen environment we have to treat the lcd_web cookbook as an artifact that can needs to be resolved via the Berksfile. Within the lcd_haproxy cookbook the Berksfile should be modified to include:

```
cookbook 'lcd_web', path: '../lcd_web'
```

- Now when `berks` is run, `lcd_web` will be referenced via the local filesystem. The `lcd_web` cookbook may not be a hard dependency in this context. This approach could suitable for integration testing, but you should have tests of both components separately as well. In this case, it may be better to generate an app and use these cookbooks with integration tests that live outside of the cookbooks themselves.
- Performing a `kitchen converge` on all these components should produce a load balancer instance which alternates between two web servers. The name of the Docker instance should be reflected in the title and page content of the web servers.

Local Cookbook Development

Community Cookbooks

What is a Community Cookbook?

- A community cookbook is a cookbook for use by anyone and is generally available at the Chef Supermarket. The public Chef Supermarket is a searchable repository for cookbooks.

  https://supermarket.chef.io

- You'll notice this URL at the top of the default `Berksfile`; this is how berks knows where to obtain dependencies from.

```
source 'https://supermarket.chef.io'
```

- You'll also notice that a `Berksfile` also references `metadata`; this is so berks knows where to get the list of dependencies from.

```
metadata
```

Private Supermarket

- A private supermarket is a supermarket URL within your control which provides cookbooks within your organization. You may prefer to obtain and distribute community cookbooks via a private supermarket because you have security or bandwidth concerns.
- In the case of a private supermarket the `Berksfile` would obtain dependencies from a supermarket other than the primary chef supermarket.

```
source 'https://supermarket.example.com'
```

Berksfile and the Supermarket

- Berkshelf can include multiple Chef Supermarket instances for dependency resolution. A `Berksfile` might contain references to multiple supermarkets.

```
source 'https://supermarket.example.com'
source 'https://supermarket.chef.io'
```

- Resolution happens in a top-down fashion. In this example, the private supermarket is consulted first and then the Chef Supermarket is chosen.

Interacting with the Supermarket

- The `knife cookbook site` commands can be used to interact with the Chef Supermarket and a private Supermarket.
- Listing all available cookbooks from the supermarket can be done with the list sub-command.

```
knife cookbook site list
```

- Alternatively you could search for something you're interested in:

```
knife cookbook site search tar
```

- The `download` sub-command will download a tar.gz file containing the cookbook.

```
knife cookbook site download tar
```

- Assuming the `cookbook_path` is pointed to a location which is an initialized git repository you can install this cookbook with the `install` sub-command.

```
knife cookbook site install tar
```

Duplicate Cookbook Names

- If you modify a community cookbook that you downloaded you can reference the local cookbook path to override the community version from being used.

```
cookbook 'tar', path: '../tar'
```

- When you `berks install` you should see:

```
Using tar (2.0.0) from source at ../tar
```

Local Cookbook Development

Managing Cookbook Dependencies

Managing Cookbook Dependencies

- Managing cookbook dependencies by manually downloading required cookbooks can present a number of challenges. Using Berkshelf makes handling dependencies much easier and simpler to keep up with.

- Berkshelf keeps the dependencies it resolves in a central location, `~/.berkshelf/cookbooks/<cookbook-version>`, which keeps your cookbooks repository free of clutter.

- Berkshelf has the ability to reference cookbooks in different ways. This includes public and private supermarkets, the local filesystem, Chef server via knife.rb, git and github. Berkshelf will resolve or make reference to these locations to either find or fetch a cookbook and related dependencies.

- The default source for dependencies is the Chef Supermarket, the metadata.rb file defines as dependencies which cookbooks the that Berkshelf needs to fetch.

Berksfile Configuration

- The default `Berksfile` is generated when you generate a cookbook and configured to reference the public Chef Supermarket via the `source` keyword. A source could be a combination of public and private supermarkets or a Chef server can be implied via the `knife.rb`.
- Anything defined as a dependency within `metadata.rb` as well as the dependences of the cookbook you depend on should be fetched.
- The `cookbook` keyword in the `Berksfile` allows you to define where you want a cookbook to be installed from or to set additional version constraints. In the example below, the local filesystem path is specified.

```
cookbook 'lcd_web', path: '../lcd_web'
cookbook 'tar', path: '../tar'
```

- You also have the ability to load cookbooks from a git repository, and include a branch or tag for specificity.

```
cookbook "my-cookbook", "~> 0.1.1", git: "https://github.com/example/my-cookbook.git", tag:
"1.2.3"
```

- Github also has a shortcut:

```
cookbook "my-cookbook", "~> 0.1.1", github: "example/my-cookbook"
```

Berksfile Configuration, Continued…

- The `group` keyword allows for a kind of categorization almost like tags that allows for berks to operate on the members of that group. A block would define a group:

```
group :test do
  cookbook "mytest-cookbook", path: "testing"
end

cookbook "mytest-cookbook", path: "testing", group: :test
```

- When berks install is run, a flag can be set to exclude or include this group.

```
berks install --except test
```

Or

```
berks install --only test
```

berks install and Berksfile.lock

- To evaluate and fetch dependencies, run berks install once your metadata has been configured.

```
berks install
```

- When `berks install` is run, a `Berksfile.lock` is created which acts like a snapshot of the versions and their dependencies. This is located with the cookbook itself and ensures that different dependent versions are not downloaded by someone else when re-evaluating the dependency tree.

- To force a re-evaluation, the `Berksfile.lock` file can be deleted and `berks` install can be rerun.

berks apply

- The version locking mechanism can be applied to entire environments, as well, by specifying an environment's JSON file. Given an development.json file:

```
{
"name": "development",
  "description": "Developemnt Environment",
  "chef_type": "environment",
  "json_class": "Chef::Environment",
  "default_attributes": {
  },
  "override_attributes": {},
  "cookbook_versions": {
    "lcd_web": "= 0.1.0"
  }
}
```

- The following command would introduce the versions which were captured during the initial `berks install` process and hard code them into an environments file.

```
berks apply development -f /tmp/development.json
```

berks apply

- The resulting versions associated are appended to the environment JSON file as a list of cookbooks and their versions.

```
"cookbook_versions":  {
    "build-essential": "= 8.0.0",
    "cpu": "= 1.0.0",
    "haproxy": "= 3.0.2",
    "lcd_haproxy": "= 0.1.0",
    "lcd_web": "= 0.1.0",
    "mingw": "= 2.0.0",
    "ohai": "= 5.0.2",
    "poise": "= 2.7.2",
    "poise-service": "= 1.4.2",
    "seven_zip": "= 2.0.2",
    "tar": "= 2.0.0",
    "windows": "= 3.0.3"
  }
```

Local Cookbook Development

Data Bags

What are Data Bags?

- Data bags are a container for information about your infrastructure that are not associated with a node. They provide an external source of information to be used in recipes and search functionality.

- A data bag is a global variable that is stored as JSON data and is accessible via a Chef server. Because the data bag is on the Chef Server, it acts as a shared resource which can be referenced by Chef clients.

- In addition to general purpose usage, data bags can also be used to encrypt sensitive information with a shared key; anyone with the shared key can decrypt the data in the data bag. This provides one method to keep secrets off of nodes.

Working with Data Bags

- Data bag commands exist to allow the management of data bags and the items they contain.

```
knife data bag create BAG [ITEM] (options)
knife data bag delete BAG [ITEM] (options)
knife data bag edit BAG ITEM (options)
knife data bag from file BAG FILE|FOLDER [FILE|FOLDER..] (options)
knife data bag list (options)
knife data bag show BAG [ITEM] (options)
```

How to Create Data Bags

- A data bag can be created with `knife` or manually as a json file.
- When using knife to create a data bag you can use the following command. Where "admins" is the name of the data bag being created and "joe" is the name of the item within the "admins" data bag. This will invoke $EDITOR to provide an interface to edit the item.

```
knife data bag create admins joe
```

- A data bag can be created with `knife` or manually as a JSON file. If you have source data it can be loaded from file.

```
knife data bag from file admins /path/to/joe.json
```

How to Create Data Bags

- As an example, joe.json might look as follows:

```
{
  "id": "joe",
  "uid": "1001",
  "gid": "developers",
  "shell": "/bin/bash",
  "comment": "Joe"
}
```

How to Edit Data Bags

- When using knife to edit a data bag you can use the following command. This provides an interface to edit the data bag with $EDITOR. When these are locally available files they can be edited directly as well.

```
knife data bag edit admins joe
```

How to Delete Data Bags

- When using knife to delete a data bag you can use the following command. You will be prompted before deleting the item to confirm.

```
knife data bag delete admins tom
```

- You can also delete the entire bag and everything in it by not specifying an element ID.

```
knife data bag delete admins
```

How to List Available Data Bags and Items

- Use the list sub-command to get a listing of the names of all the data bags available.

```
knife data bag list
```

- Use the show sub-command to list available items within a bag where "admins," in the example below, is the name of a data bag.

```
knife data bag show admins
```

- Use the show sub-command along with an item name to output that item within a bag where "admins," in the example below, is the name of a data bag and "joe" is the name of an item within that bag.

```
knife data bag show admins joe
```

How to Search Data Bags

- Searching data bags can be accomplished with `knife search`. Knife search operates on an index; when searching a data bag, the name of the bag is the index you are searching.
- Searches contain both a key and a pattern to search; in the example below, the key is "id" and the search pattern is "tom".

```
knife search admins "id:tom"
```

- You can also use operators to include or exclude results:

```
knife search admins "NOT id:tom"
```

```
knife search admins "gid:developers OR gid:administrators"
```

- Wildcards

```
knife search admins "*:*"
```

```
knife search admins "gid:admin*"
```

Using Data Bags in Recipes

- The recipe DSL provides methods to interact with data bags called `data_bag` and `data_bag_item`. Listed below is an example of usage for each method.

```
data_bag('admins')

data_bag_item('admins', 'joe')
```

- These can be assigned as variables within recipe code to be consumed in resources.

```
admins = data_bag('admins')

admins.each do |login|
  admin = data_bag_item('admins', login)
  group admin['gid']
end
```

Using Data Bags in Recipes

- Since data bags are a kind of index that can be searched, they can also be used with the recipe DSL search method. Items returned by a search can be used as if they were a hash.

```
admins = []
search(:admins, "gid:administrators").each do |admin|
  group admin['gid']
  login = admin["id"]
  admins << login
  home = "/home/#{login}

  user(login) do
    uid        admin['uid']
    gid        admin['gid']
    shell      admin['shell']
    comment    admin['comment']
    home       home
    manage_home true
  end
end
```

Local Cookbook Development

Chef Vault

What is Chef Vault?

- Vault allows the secure distribution of secrets as an alternative to encrypted data bags. Vault allows limiting which users and nodes have access to secrets.

- Vault creates an encrypted data bag which is symmetrically encrypted using a random secret. A vault has a list of administrative users and clients. The shared secret is asymmetrically encrypted for each of the administrators and clients using their public key generating a separate copy for each user or client.

- By comparison, encrypted data bags leave key distribution up to the user and a shared secret needs to be present on the node or available via URI.

- The vault use case is when enhanced controls for managing secrets are needed.

Local Cookbook Development

Search

Search

- Using search allows you to facilitate more dynamic configurations by asking questions about aspects of your infrastructure.

- Helps keep data and configuration separate.

- Search works on one of the following indexes
  - client
  - environment
  - node
  - role
  - data bag name

Knife Search

- The `knife search` command is the ChefDK tool which can be used to search from the command line.

- Searches contain the index to be searched and a key with a pattern to search. Where INDEX is one of client, environment, node, role or the name of a data bag. When the index is not specified the default is to search the node index.

```
knife search INDEX SEARCH_QUERY
```

- The `SEARCH_QUERY` component is comprised of a key and pattern. Where key is a field name that is found in the JSON description of an indexable object on the Chef server (a role, node, client, environment, or data bag) and search_pattern defines what will be searched for, using one of the following search patterns: exact, wildcard, range, or fuzzy matching.

Knife Search

- The data being searched can be a nested JSON data structure which may be multiple layers deep. These nested fields become flattened for search. That means the nested elements become valid keys for search.

```
knife search node "mtu:1500"
```

- Operators are also allowed in search queries `AND`, `OR` and `NOT` to create inclusions or exclusions in your search pattern. You can also use basic wildcards like "*" to glob and "?" to replace a single character in a search.

```
knife search admins "gid:developers AND id:j*"
```

- The ability to search nested attributes is available via "-a" switch and the main and nested attributes value.

```
knife search node <query_to_run> -a <main_attribute>.<nested_attribute>

knife search role "*:*" -a override_attributes.haproxy.app_servers_role
```

How to Search Data Bags

- Searching data bags can be accomplished with `knife search`. Knife search operates on an index, when searching a data bag, the name of the bag is the index you are searching.
- Searches contain both a key and a pattern to search, in the example below the key is "id" and the search pattern is "tom".

```
knife search admins "id:tom"
```

- You can also use operators to include or exclude results:

```
knife search admins "NOT id:tom"
```

```
knife search admins "gid:developers OR gid:administrators"
```

- Wildcards

```
knife search admins "*:*"
```

```
knife search admins "gid:admin*"
```

Using Search in Recipes

- Since data bags are a kind of index that can be searched, they can also be used with the recipe DSL search method. Items returned by a search can be used as if they were a hash.

```
admins = []
search(:admins, "gid:administrators").each do |admin|

  group admin['gid']

  login = admin["id"]
  admins << login
  home = "/home/#{login}

  user(login) do
    uid        admin['uid']
    gid        admin['gid']
    shell      admin['shell']
    comment    admin['comment']
    home       home
    manage_home true
  end
end
```