Configuration Management tool [CHEF]

Infrastructure as a CODE.

Overview of Configuration Management Tools

Even though each *CM* tool has its own terms, philosophy and ecosystem, they typically share many characteristics and have similar concepts.

Most configuration management tools use a controller/master and node/agent model. Essentially, the controller directs the configuration of the nodes, based on a series of instructions or *tasks* defined in your provisioning scripts.

Below you can find the most common features present in most configuration management tools for servers:

**Automation Framework**

Each CM tool provides a specific syntax and a set of features that you can use to write provisioning scripts. Most tools will have features that make their language similar to conventional programming languages, but in a simplified way. Variables, loops, and conditionals are common features provided to facilitate the creation of more versatile provisioning scripts.

**Idempotent Behavior**

Configuration management tools keep track of the state of resources in order to avoid repeating tasks that were executed before. If a package was already installed, the tool won't try to install it again. The objective is that after each provisioning run the system reaches (or keeps) the desired state, even if you run it multiple times. This is what characterizes these tools as having an *idempotent behavior*. This behavior is not necessarily enforced in all cases, though.

**System Facts**

Configuration management tools usually provide detailed information about the system being provisioned. This data is available through global variables, known as *facts*. They include things like network interfaces, IP addresses, operating system, and distribution. Each tool will provide a different set of *facts*. They can be used to make provisioning scripts and templates more adaptive for multiple systems. **In chef we call System discovery tool ohai.**

**Templating System**

Most CM tools will provide a built-in templating system that can be used to facilitate setting up configuration files and services. Templates usually support variables, loops, and conditionals that can be used to maximise versatility. For instance, you can use a template to easily set up a new virtual host within Apache, while reusing the same template for multiple server installations. Instead of having only hard-coded, static values, a template should contain placeholders for values that can change from host to host, such as NameServer and DocumentRoot**. In chef we use embedded ruby as Templating system.**

**Extensibility**

Even though provisioning scripts can be very specialized for the needs and demands of a particular server, there are many cases when you have similar server setups or parts of a setup that could be shared between multiple servers. Most provisioning tools will provide ways in which you can easily reuse and share smaller chunks of your provisioning setup as modules or plugins.

Third-party modules and plugins are often easy to find on the Internet, specially for common server setups like installing a PHP web server. CM tools tend to have a strong community built around them and users are encouraged to share their custom extensions. Using extensions provided by other users can save you a lot of time, while also serving as an excellent way of learning how other users solved common problems using your tool of choice. **In chef we are having extensions like Chef super market, Ruby gems, Knife plugins etc.**

**Three major components of chef:**

1. **Chef development kit**- To develop and test your infrastructure. Automating code locally.
2. **Chef Client:** A chef client is going to run on each node to securely communicate with the chef server. Chef client can use instruction in the form of recipes or cookbooks to bring a node into the desired state.
   1. We have to run chef client as root, so we can bring the system to desired state.
   2. Chef client runs with two different mode : **default mode**: chef-client's default mode attempts to contact a Chef Server and ask it for the recipes to run(**Run list**) for the given node AND **locally**: (chef-client - - local-mode hello.rb) .
3. **Chef Server:** Chef server is going to act as central artifact repository for all your information used in code that actually provision infrastructure in the nodes.

**Resources(Fundamental building block):** This is the smallest configurable pieces that we can use with chef.

A resource is a statement of configuration policy that:

* Describes the desired state for a configuration item
* Declares the steps needed to bring that item to the desired state
* Specifies a resource type—such as package, template, or service
* Lists additional details (also known as resource properties), as necessary
* Are grouped into recipes, which describe working configurations

Where a resource represents a piece of the system (and its desired state), a provider defines the steps that are needed to bring that piece of the system from its current state into the desired state.

The Chef::Platform class maps providers to platforms (and platform versions). At the beginning of every Chef Client run, Ohai verifies the platform and platform\_version attributes on each node. The Chef Client then uses those values to identify the correct provider, build an instance of that provider, identify the current state of the resource, do the specified action, and then mark the resource as updated (if changes were made).

For example:

directory '/tmp/folder' **do**

owner 'root'

group 'root'

mode '0755'

action :create

**end**

The Chef Client will look up the provider for the directory resource, which happens to be Chef::Provider::Directory, call load\_current\_resource to create a directory["/tmp/folder"] resource, and then, based on the current state of the directory, do the specified action, which in this case is to create a directory called /tmp/folder. If the directory already exists, nothing will happen. If the directory was changed in any way, the resource is marked as updated.

## Resource Syntax

A resource is a Ruby block with four components: **a type, a name, one (or more) properties (with values), and one (or more) actions**. The syntax for a resource is like this:

type 'name' **do**

attribute 'value'

action :type\_of\_action

**end**

Every resource has its own set of actions and properties. Most properties have default values. Some properties are available to all resources, for example those used to send notifications to other resources and guards that help ensure that some resources are idempotent.

For example, a resource that is used to install a tar.gz package for version 1.16.1 may look something like this:

package 'tar' **do**

version '1.16.1'

action :install

**end**

All actions have a default value. Only non-default behaviors of actions and properties need to be specified. For example, the **package** resource’s default action is :install and the name of the package defaults to the name of the resource. Therefore, it is possible to write a resource block that installs the latest tar.gz package like this:

package 'tar'

and a resource block that installs a tar.gz package for version 1.6.1 like this:

package 'tar' **do**

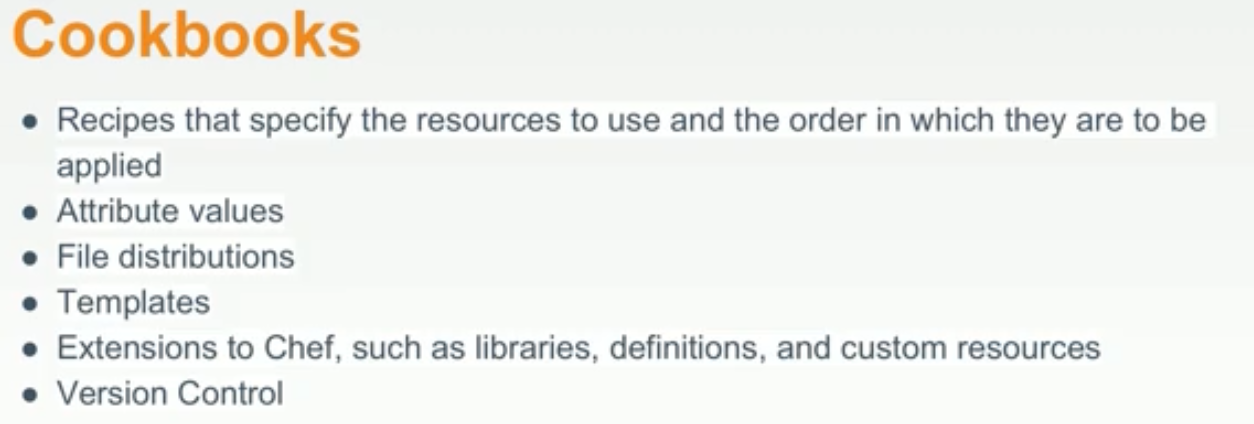
version '1.16.1'

**end**

In both cases, the Chef Client will use the default action (:install) to install the tar package.

[ Recipe is collection of resources.]

[ Cookbook is collection of Recipes]



Cookbook Component:

Components on “workstation” cookbook



Berksfile: Used for managing dependencies for our cookbook.

Chefignore: Very similar to git ignore, Directories you would like to exclude from your chef server.

metadata.rb : will contain information like name , version, who is responsible etc of the cookbook. Also we will store dependencies inside metadata.rb file.

recipes : this will contain all the recipes inside a cookbook. One default recipe will be contained with every cookbook we create.

Spec and test: These are local directories used for testing locally.

Spec is for unit testing using common unit testing framework call arspec for ruby.

Test is for integration testing using framework call test kitchen.

**CHEF CLIENT:**

Using chef-client we can run recipes from one or more cookbooks.

Recipes in a run list are comma separated, runlist should not contain spaces.

Example:

Running one recipe: chef-client -z -r "recipe[apache::server]"

Running a list of recipe: chef-client -z -r "recipe[apache::server], recipe[workstation::setup]"

**INCLUDE\_RECIPE:**

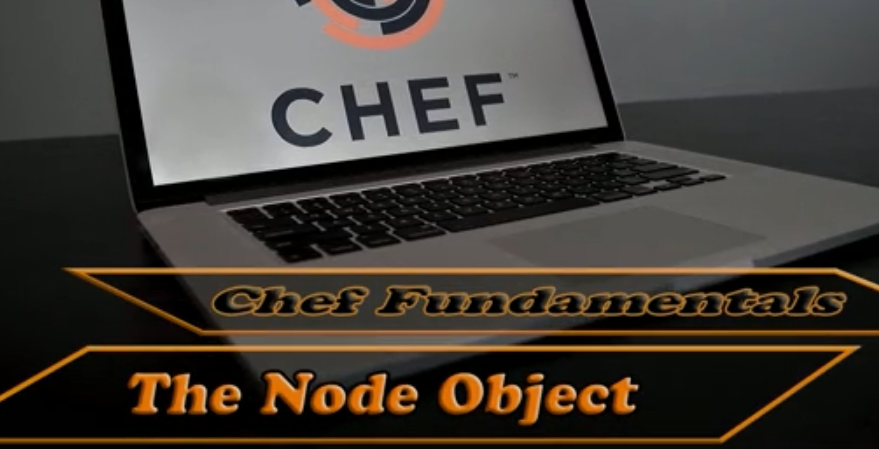
**It is a method to Call recipe/recipes inside another recipe**

Include\_recipe is used to call one or more recipes inside a cookbook from another recipe.

Read include recipe doc for more details.

**Ohai:**

**It is a system dis tool used by chef.**



**Read doc for more details about ohai.**

**Accessing node attributes:**

**We can access attributes of a node as below:**

**Normally:**

node[ ‘ipaddress’ ]

node[ ‘hostname’ ]

**Inside a string: using string interpolation in ruby.**

file '/etc/motd' do

content "This sever is property of Biron Boro

HOSTNAME: #{node['hostname']}

IP ADDRESS: #{node['ipaddress']}

CPU: #{node['cpu']['0']['mhz']}

MEMORY: #{node['memory']['total']}

"

**Template:**

We have studied about resources like file resource, package resource and service resource.

A template is a cookbook component, cookbook has template as an additional feature.

Templet resource is much more mature way to handle files that exist on a node.

In case of big application, it is difficult to manage code inside content property of a file resource.

In that case we can write our code in native format using template resource. That will be an embedded ruby file which will allow us to import node attributes inside any native code format by use of erb tags.

**Understanding Embedded ruby:**

An Embedded Ruby (ERB) template allows Ruby

code to be embedded inside a text file within

specially formatted tags.

Ruby code can be embedded using expressions

and statements.

<https://docs.chef.io/templates.html#variables>

**Text Within an ERB Template**

**<% if (50 + 50) == 100 %>**

**50 + 50 = <%= 50 + 50 %>**

**<% else %>**

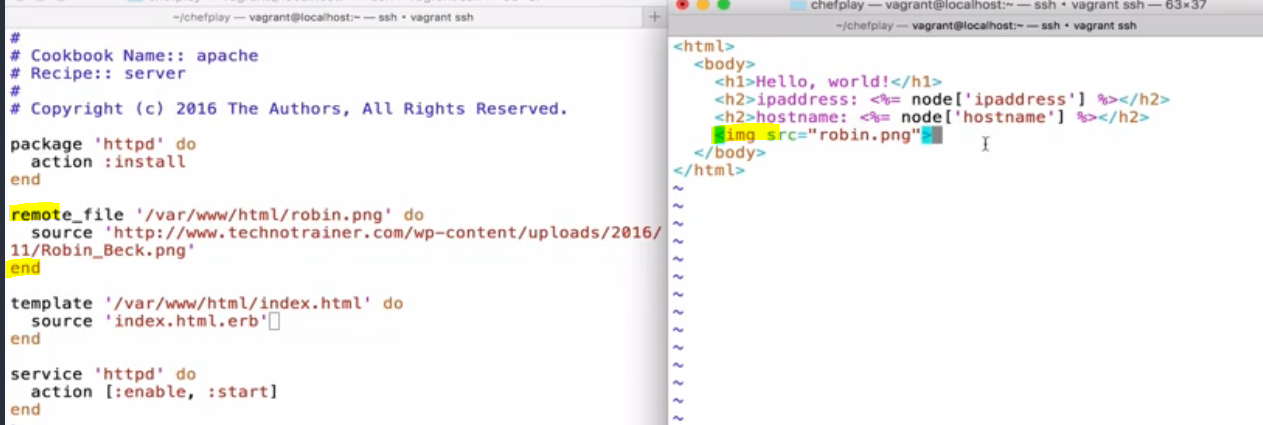
**At some point all of MATH I learned in school changed.**

**<% end %>**

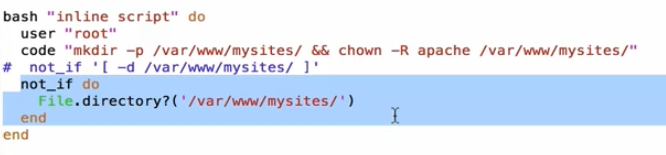
Each ERB tag has a beginning tag and a matched ending tag.

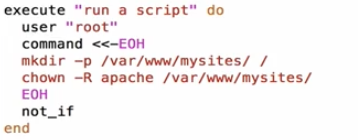
Other Common Resources in Chef:

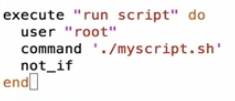
1. cookbook\_file: it is used for purely static content.
2. remote\_file: it is used to download file from remote source.



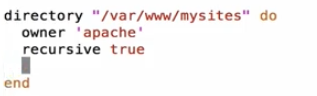
1. execute: this is used to execute some script or some content of a script.







1. directory Resource:



1. Users and Groups: Used to create and user id and Groups.
2. Sending and receiving notifications: Notification is a common property to all resources. All resources have the ability to notify or subscribe another resource if its state changes in someway

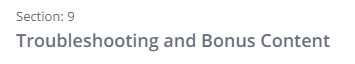
Here in both cases httpd service will get restarted only if there is some modification or changes in the template file.

**Case1: notifies:**



**Case2: subscribes:**





**Chef comes with various embedded tool installed inside chef development kit.**

**Some of embedded installed tool: ruby-🡪 to use this tool use command “ chef exec ruby –help “**

<https://docs.chef.io/ruby.html>

<https://docs.chef.io/debug.html>

**Test Kitchen: Test kitchen is a frame work that allows us to deploy our code in a isolated environment where it can be tested.**

<https://docs.chef.io/kitchen.html>

<https://inscpect.io> **--🡪 inscpect is to write test cases.**

1. **Make sure Kitchen is installed in the box where we are writing chef code.**
2. **Now we have to provision the virtual box/ ec2 instance/ docker container to be configured by Kitchen for testing purpose.**

**To do date:**

1. **Provide all vm/ec2 instance details inside cookbook\_name/.kitchen.yml file.**

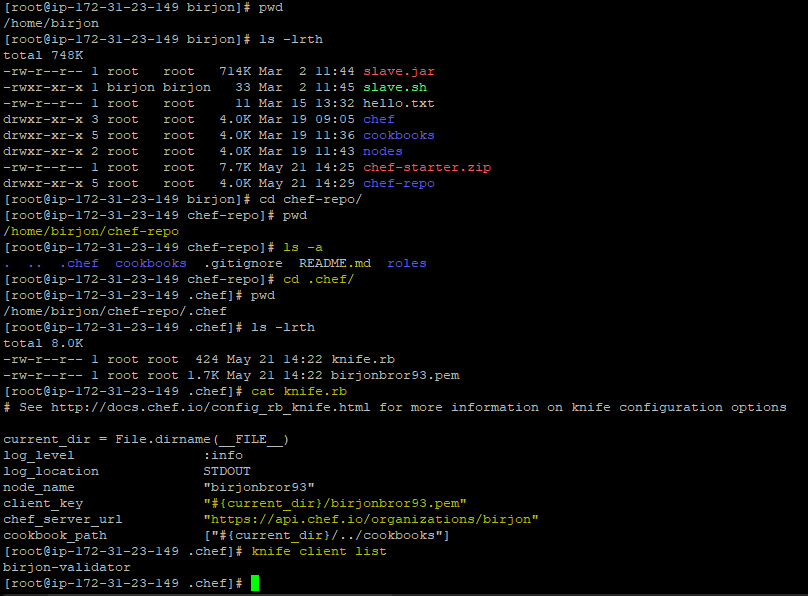
**Run below commands inside a particular cookbook directory that you want to test**

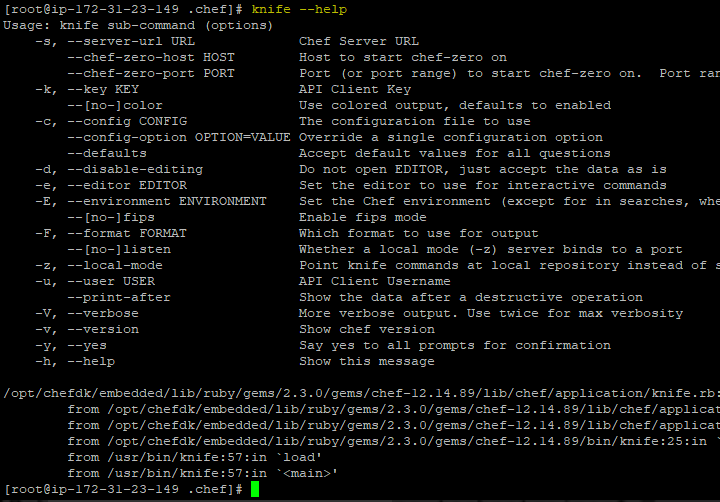
1. **Then run “kitchen create” to setup the virtual env.**
2. **Then run “kitchen converge” to converge the cookbook from local to virtual env.**
3. **Then “kitchen verify” to apply all the test case that we have written inside test directory.**
4. **We can log in to the vm/ec2 instance by running command “Kitchen login”**
5. **“Kitchen --help” 🡪 all the useful commands for kitchen**
6. **“kitchen list” will list all the vm with other useful details like**

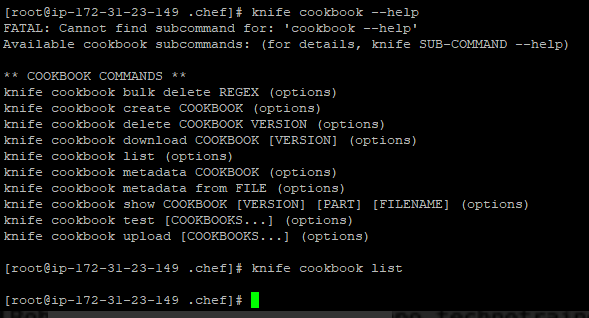
**Chef spec ( Unit test ):**

**Chef Server:**

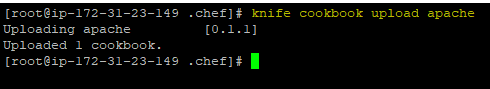
1. **Create an account in chef server (**<https://manage.chef.io/>**)**
2. **Create organization inside Administration tab.**
3. **Starter kit 🡪 Download starter kit**
4. **Place the chef-starter.zip file where the chef-dk is installed and unzip the file.**
5. **After unzip of the file we will get one chef-repo (refer below screen shots)**
6. **Post that we can upload cookbooks from the box( chef\_dk )**







**Knife command is used to communication with chef server from local workspace.**



**What bootstrapping in Chef:**

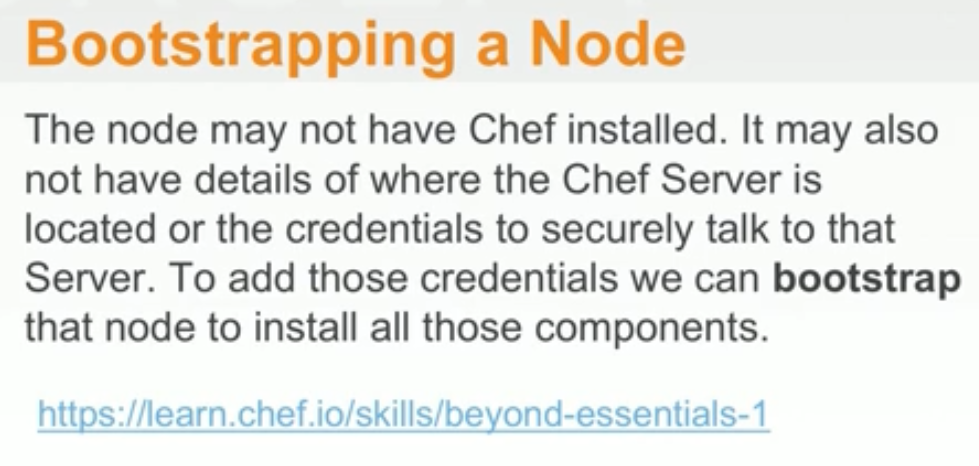
**Let’s talk about attaching node to the chef server.**

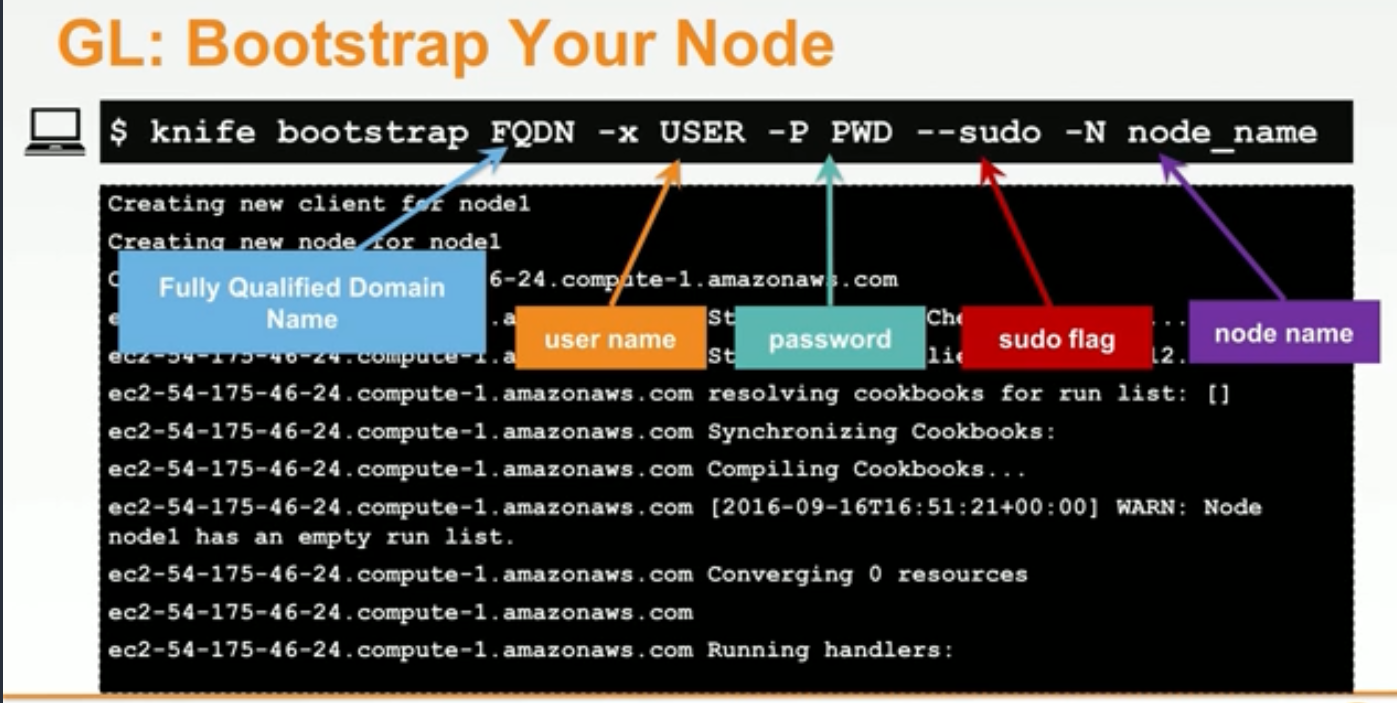
**This the process known as bootstrapping,**

**Bootstrapping is generally getting a node ready to configure with chef.**

**The node that you want to configure with chef might not have chef installed, it may not know how to communicate with the chef server, and it does not have really a list of cookbooks that needs to run off the back.**

**The process of filling up three steps is bootstrapping.**





**To install chef client in Web1 , Web2 and load balancer box :**

**curl -L https://omnitruck.chef.io/install.sh | sudo bash -s -- -v 12.21.4**

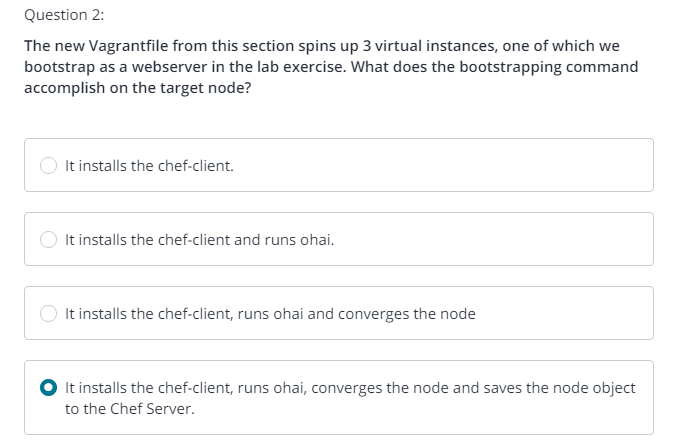
**Bootstrap command for AWS env:**

knife bootstrap ip\_address\_of\_node - - ssh-port 443 - - ssh-user ec2-user - - sudo - - indentity-file /home/birjon/chef-repo/key.pem -N Web1

**Some Useful Knife commands:**

1. knife cookbook upload apache**: To upload cookbook to chef server. (apache is name of a cookbook)**
2. knife cookbook list **: This command is used to list all the cookbook in the chef server.**
3. Knife node list **: This command is used to list all the nodes associated with the chef server.**
4. knife node show Web1 **: To show all details of a node (Web1 is name of node)**
5. knife node run\_list **add** Web1 "recipe[workstation],recipe[apache]" **: to add run list to node.**
6. sudo chef-client **: Running this command in a node will execute all the recipes in the run list added for that particular node.**
7. Knife node run\_list **set** Web1 “recipe[mychef-client],recipe[workstation],recipe[apache]”

**This command will replace the existing runlist of the node Web1**



**https://docs.chef.io/attributes.html**

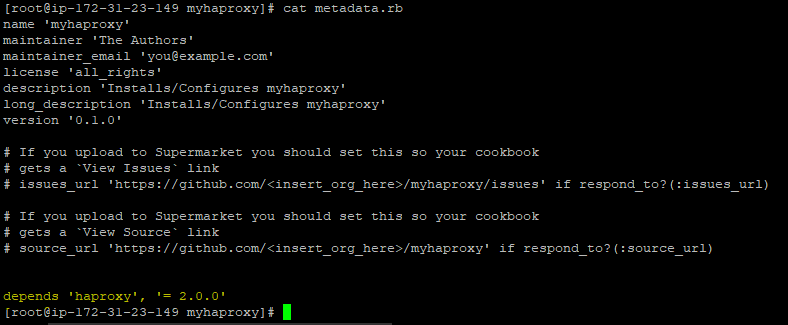
**Chef Super market / Community cookbook (https://supermarket.chef.io/):**

Chef super market provide us inbuilt cookbooks. We will use those cookbook as dependency library for our cookbook. We are not going to change any attribute inside the inbuilt cookbook/library. We will call/import the inbuilt cookbook as library inside a wrapper cookbook, where we will use only those attributes from the inbuilt cookbook and modify those attribute values according to our requirement inside wrapper cookbook itself.

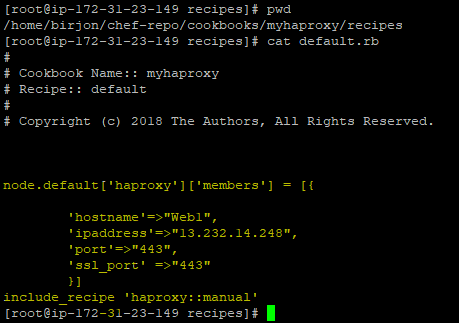
Managing Cookbook with **Berkshelf**:

Step 1: Create one Wrapper cookbook: chef generate cookbook cookbooks/myhaproxy

Step 2: Edit metadata.rb

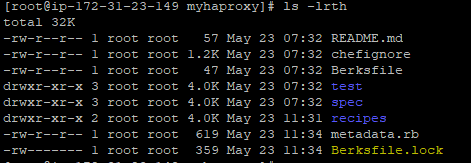


Step 3: Edit the default.rb file inside the Cookbook



Step 4: berks install 🡪 to install all the dependencies locally.

Once we do berks install then one berkshelf.lock file get generated.



This file contains dependency details of the cookbook from super market (haproxy in this case).

Step 4: berks upload 🡪 to upload cookbook to chef server.

We can use “berks upload --force” to update/overwrite the chef server with same version of the cookbook for some small changes in the cookbook, where we don’t want to increase our version/patch number of the cookbook.

Step 5: expand a new vm/instance for load-balancer.

Step 6: bootstrap the new node:

Knife bootstrap ip\_address –ssh-port 443 –ssh-user ec2-user –sudo –indentity-file key.pem -N load-balancer

Step 7: add run list for that node

Knife node run\_list add load-balancer “recipe[myhaproxy]”

Step 8: Run chef-client in the load balancer node.

**Verifying node Data with knife: Knife node show –help**

**Knife node list**

**Chef-client on a schedule: chef-client cookbook from supermarket**

<https://supermarket.chef.io/cookbooks/chef-client>

**Roles:**

**Way of centralizing recipes or run\_list for similar nodes.**

**Roles are a way to describe a custom run list or a set of cookbooks and recipes that should be running on a particular type of node.**

A role is a way to define certain patterns and processes that exist across nodes in an organization as belonging to a single job function. Each role consists of zero (or more) attributes and a run-list. Each node can have zero (or more) roles assigned to it. When a role is run against a node, the configuration details of that node are compared against the attributes of the role, and then the contents of that role’s run-list are applied to the node’s configuration details. When a chef-client runs, it merges its own attributes and run-lists with those contained within each assigned role.

Rather than adding/editing run list for number of nodes separately or one at a time.

We can use role to manage run list for number of nodes.

**Demo Creating and Assigning Roles:**

**Knife role –help, will show how to manage roles on your chef server.**

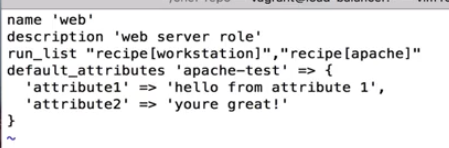
**Knife role list, knife role show is some of use full commands.**

**There are different ways of creating roles:**

1. **From Command line:** knife role create role\_name
2. **From a file: With the starter kit we will get one “roles/” Directory.**

**Inside this directory we can create one file named “file\_name.rb” and inside this file we will define roles and run\_list associated with that role.**

**For example : roles/web.rb**



**Then run command : knife role from file roles/web.rb**

**Now we can upload the roles to the chef server using run\_list command.**

**Remember roles are nothing than a custom run\_list.**

**knife node run\_list set web1 “role[web]”**

**post the node converge to its desired state after running chef-client .**

**the role filed will be populated the role that we have added to the node.**

**Note: We can write the role.rb file with ruby dsl or with pure json.**

1. **We can create role directly from chef server console.**

**Converge nodes using ‘knife ssh’ command :**

knife ssh node\_ipaddress ‘sudo chef-client’ –manual-list –ssh-port 22 –ssh-user ec2-user --identity-file key.pem

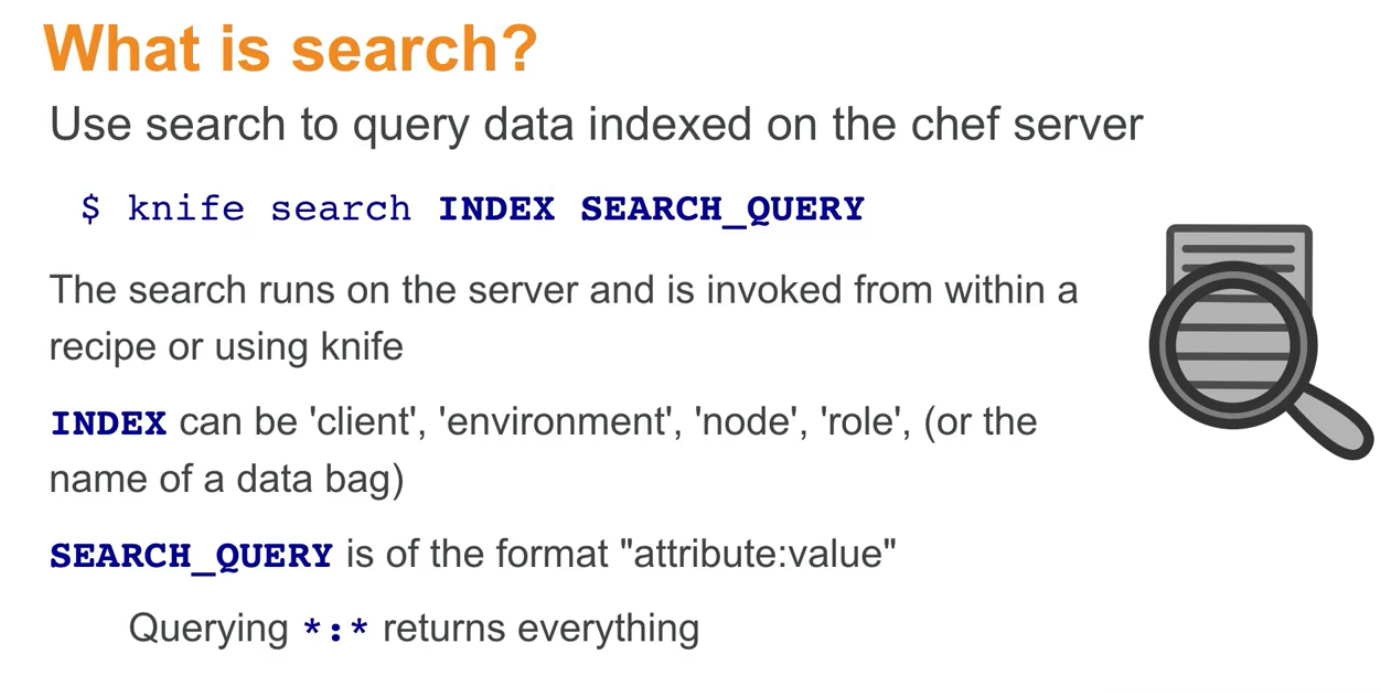
We can use search patter as well rather than passing node information manually.

knife ssh “**\*:\***” -x user\_name -p password ‘sudo chef-client’

knife ssh “**role:web\***” -x user\_name -p password ‘sudo chef-client’

**Indexing and search:**

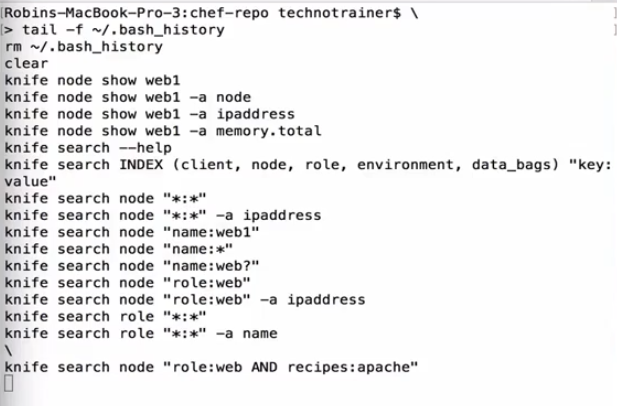
**Note: Every time chef client run, it save node object back to chef server.**



**Node index: this index indexes all the node object for your infrastructure, any node that is attached to a particular chef server organization, we will have its node object saved back to its chef server.**

**Role index: We keep track of all nodes that have a particular role and values associated with nodes.**

**Environment index: We keep track of all nodes that to which env the node is attached and values associated with nodes.**



**Dynamic HA Proxy cookbook recipe, using search function.**



**For Cloud:**

# ~/chef-repo/cookbooks/myhaproxy/recipes/default.rb

#

# Cookbook Name:: myhaproxy

# Recipe:: default

#

# Copyright (c) 2016 The Authors, All Rights Reserved.

#

# This recipe is for using the public hostname and public ipaddress

# on any cloud instance, such as AWS, Azure or GCP

all\_web\_nodes = search('node',"role:web")

members = []

all\_web\_nodes.each do |web\_node|

member = {

'hostname' => web\_node['cloud']['public\_hostname'],

'ipaddress' => web\_node['cloud']['public\_ipv4'],

'port' => 80,

'ssl\_port' => 80

}

members.push(member)

end

node.default['haproxy']['members'] = members

include\_recipe "haproxy::manual"

Environment: Control your Deployment.

Each organization is going to start with a single environment called the \_default environment, which can not be modified or deleted . Therefor you must create custom environment to define your organizations work flow.

In a custom environment we can define cookbook restriction rule ( which version of cookbook a node can converge).

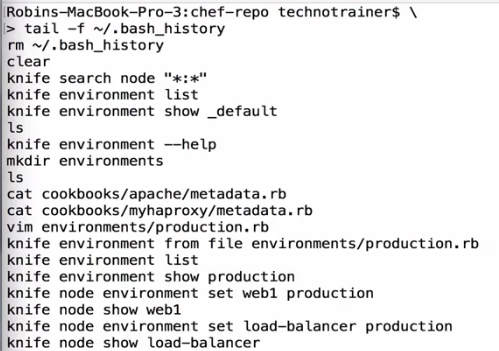
Like Role, we should create environment and attached to required nodes.

Like Role, We can create an environment from

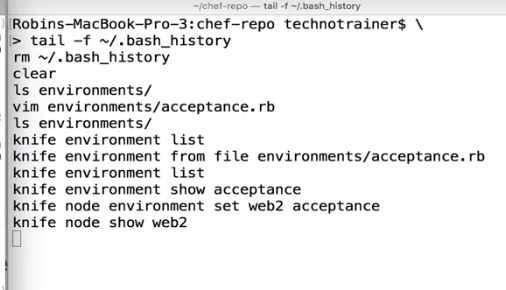
1. a ruby/json file: knife environment from file environments/production.rb
2. using knife command: “ knife environment create env\_name “
3. from chef server console.

Creating and managing production environment:

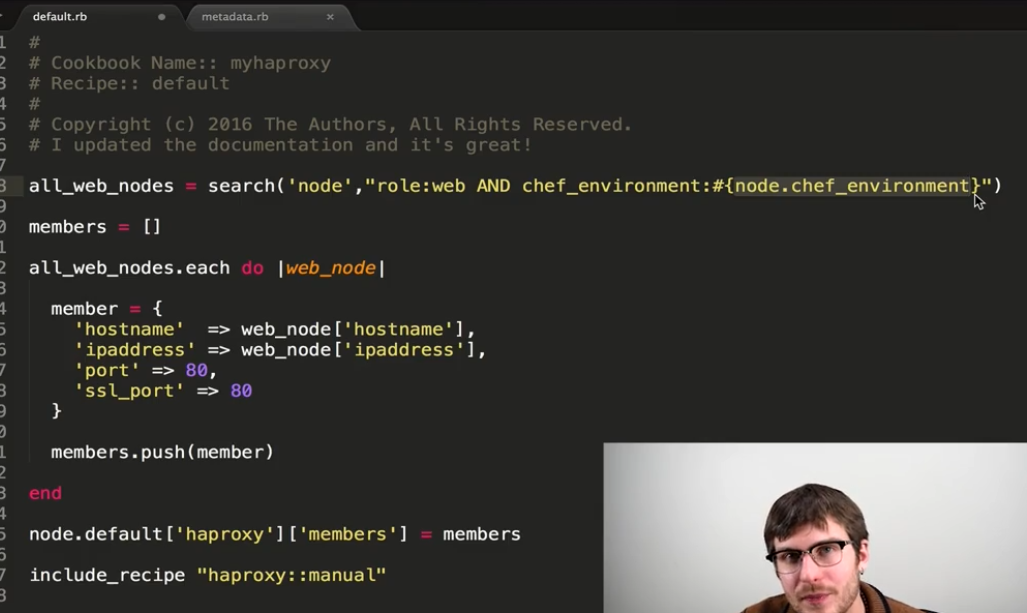
<https://docs.chef.io/environments.html#create-environments>



Acceptance environment:



Utilizing environment inside a Recipes:



Update the myhaproxy cookbook version at metadata.rb file.

Then run “berks install”

And then “berks upload”

Currently the load balancer node(Which is in production environment) is restricted to use a specific version of myhaproxy cookbook.

Update the environment/production.rb file with latest cookbook version of myhaproxy.

Then , 🡪 knife environment from file environment/production.rb

* Knife environment show production.

Ssh to load balancer node.

* Sudo chef-client

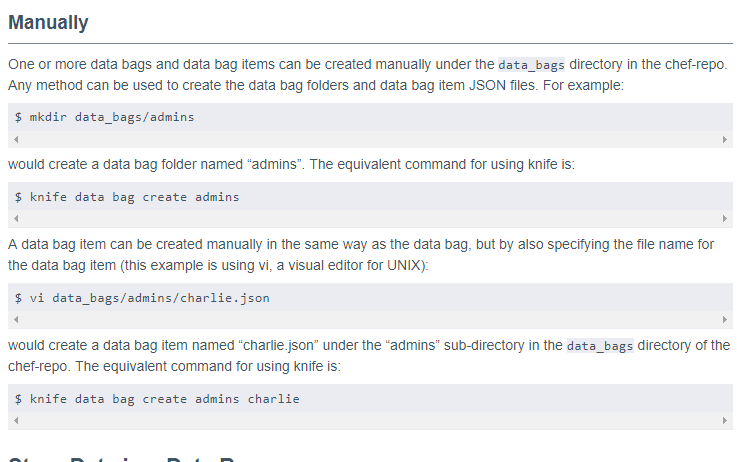
Data Bags:

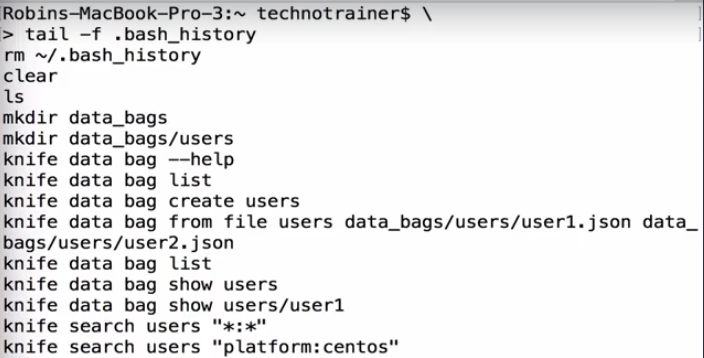
Organizing custom data bag.

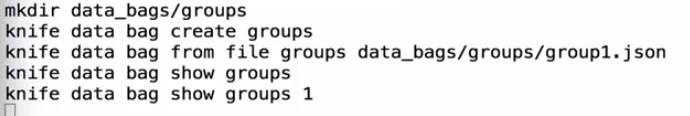
A data bag is a global variable that is stored as JSON data and is accessible from a Chef server. A data bag is indexed for searching and can be loaded by a recipe or accessed during a search.

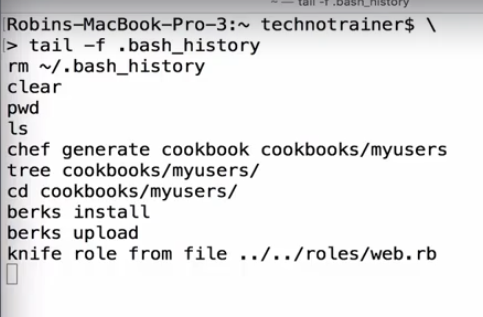
Creating data bag:

1. Manually:









Encrypting data bag using secret-file :

