**What to expect from this document:**

This document talks about the basic constructs in Chef and help them explain in simple terms. Also, explains with simple examples how to use these basic constructs for creating your own cookbook.

**Attributes:**

An attribute is a specific detail about a node. Attributes are used by the chef-client to understand:

* The current state of the node
* What the state of the node was at the end of the previous chef-client run
* What the state of the node should be at the end of the current chef-client run

Attributes are provided to the chef-client from the following locations:

* Nodes (collected by Ohai at the start of each chef-client run)
* Attribute files (in cookbooks)
* Recipes (in cookbooks)
* Environments
* Roles

After the node object is rebuilt, all of attributes are compared, and then the node is updated based on attribute precedence. At the end of every chef-client run, the node object that defines the current state of the node is uploaded to the Chef server so that it can be indexed for search

Attributes defined in a dependent cookbooks can also be accessed in the recipe.

The Attributes file can be roughly thought of as analogous to the Constants file in java

**Roles:**

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.

In your organization, if your infrastructure grows to meet the demands of higher traffic, there are likely to be multiple, redundant servers that all perform the same basic tasks. For instance, these might be web servers that a load balancer passes requests to. They would all have the same basic configuration and could be said to each satisfy the same "role".

The easiest way of creating a role is using the knife command

knife role create <roleName>

This creates a json corresponding to the role. You can add recipes to the runlist by editing the json.

In order to apply this role to the node, use below command

knife node edit ip-10-230-29-161.ec2.internal

Roles can inherit from other roles. Runlist in the roles can include a role.

**Environments:**

An environment is a way to map an organization’s real-life workflow to what can be configured and managed when using Chef server.

An environment is simply a designation meant to help an administrator know what stage of the production process a server is a part of. **Each server can be part of exactly one environment**.

By default, an environment called "\_default" is created. Each node will be placed into this environment unless another environment is specified.

An environment can be created in five different ways:

* Creating a Ruby file in the environments sub-directory of the chef-repo and then pushing it to the Chef server
* Creating a JSON file directly in the chef-repo and then pushing it to the Chef server
* Using Knife
* Using the Chef Manage web user interface
* Using the Chef server REST API

Each **environment** should have some specific set of attributes. cookbook\_versions is one of them. One of the **major advantages** of incorporating environments into your system is that you can **specify version constraints for the cookbooks**, and recipes that are deployed. Environments can be used to change the versions of cookbooks to be applied.

The knife tool can generate the template of an environment file by typing

knife environment create <env\_name>

A node is considered to be associated with an environment when the chef\_environment attribute is set. You can edit the node to add the environment to it.

Search command can be used to search nodes based on the environment attribute.  **knife search node "chef\_environment:dev"**

**Resources:**

A resource is a statement of configuration policy. Resources describe all the components in your network. Recipes group resources together and describe working configurations. Add recipes to a run-list to describe the desired state for every node to which that run-list is assigned. Cookbooks are collections of recipes and are stored on the Chef server.

A resource is a Ruby block with four components: a type, a name, one (or more) attributes (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 attributes**

Chef client run workflow:

Chef client run 🡪 identify the platform based on the values of ‘platform’ and ‘platform\_version’ attributes returned by Ohai🡪use this values to identify the correct provider🡪build instance of that provider 🡪 identify the current state of the resource 🡪 do the specified action 🡪 mark the resource as updated (if changes were made)

**Data Bags:**

A data bag is a global variable that is stored as JSON data and is accessible from a Chef server. It contains databag items. Chef-repo will have the data bags dir. Create the folder with data bag name. Inside the folder, create a json file with the data bag item name.

knife create data bag <data\_bag\_name> knife data bag from file <data\_bag\_name> <item\_name.json>

this command will read the item file created and upload it on the chef server

**Templates:**

They can be used to generate configuration files. To use a template, two things must happen:

* A template resource must be added to a recipe
* An Embedded Ruby (ERB) template must be added to a cookbook

A very basic template e.g is as follows:

template "/tmp/foo" do

  source "**foo.erb**"

  variables({

    :x\_men => "are keen"

  })

end

**foo.erb** file should be present in the **cookbook/attributes/default** directory. The /tmp/foo directories should be pre-created

e.g contents of the foo.erb file are

The node <%= node[:fqdn] %> thinks the x-men <%= @x\_men %>

It can contain ruby expressions which are evaluated during a chef client run.

**Anatomy of a Chef client run:**

It consists of 2 phases:

* **Compile:**
* Load all attributes files. All attributes are within the scope of the node
* Load all cookbooks/recipes
* Read each recipe and create a resource collection out of it

e.g. resource\_collection = [package ["apache2"], service ["apache2"]]

* **Execute phase**:
* This phase involves the actual execution of the resource collection.
* Each run of the chef client is idempotent.
* E.g. package["apache2"]

Is it in the appropriate state i.e. is the resource installed?

Yes - do nothing

No - Take the action to bring the resource in line with the desired state

This happens for each resource.

**Using Attributes in recipes:**

An attribute can be defined in a cookbook (or a recipe) and then used to override the default settings on a node.

A Chef attribute can be seen as a variable that:

1. gets initialized to a default value in cookbooks/mycookbook/attributes/default.rb
2. gets used in cookbook recipes such as cookbooks/mycookbook/recipes/default.rb
3. can be overridden at either the role or the node level

An attribute file is located in the attributes/ sub-directory for a cookbook. When a cookbook is run against a node, the attributes contained in all attribute files are evaluated in the context of the node object.

sample attributes/default.rb file

node.default[:dir] = "/tmp/test"

node.default["data\_dir"] = "/tmp/testdata"

This can be also written as below:

default[:dir] = "/tmp/test"

default["data\_dir"] = "/tmp/testdata"

Attributes defined in default.rb can be used in the recipes as below:

directory "#{node["data\_dir"]}" do

  action :create

  owner "ubuntu"

  mode "0777"

end

**The attributes defined in the default.rb are accessible as node attributes in the recipe**

**Use of attributes – best practice:**

Wherever hardcoded values are used in recipes, it's almost always better to use an attribute instead, and set the default value of the attribute to that hardcoded value, which can be then overridden as needed.

**Knife Plugins:**

There are different knife plugins that are available. Below is an example

knife rackspace server create -r 'role[webserver]' -N nodeName

This is a knife rackspace plugin which will spawn a new vm and bootstrap the node and apply the role

**Notifies Syntax :**

The basic syntax of a notifies notification is:

resource "name" **do**

notifies :notification, "resource[name]", :timer

**end**

**LWRPs (Light Weight Resource Provider):**

Recipe should describe the state of our server but not the action taken, and the action logic need to be separate into provider. That is the reason why in most cases, we need to create a custom resource and provider.

In order to write custom resources and resource providers, we need to have good knowledge about Ruby. The resource class should be extended from the Chef::Resource and the provider class should be extended from the Chef::Provider. And should know about all the constructs in Ruby viz. Class, inheritance, etc. This is the **heavy way** of creating a custom provider.

Chef provides a DSL to ease this and it’s called LWRP DSL.

* Resource file is created in the cookbook/resources folder (e.g. default.rb). This file contains the actions and attributes that can be used in the recipe of the custom resource.

e.g.

actions :create, :remove

attribute :title, kind\_of: String, default: "World"

attribute :path, kind\_of: String, default: "/tmp/greeting.txt"

* The provider has to implement each action defined in the resource e.g.

action :create do

...

end

action :remove do

...

end

* In this implementation, we can use the different platform resources available in Chef like file, package etc. The values of attributes specified in the recipe are accessible here using the ‘new\_resource’ variable.

Recipe should describe the state of our server but not the action taken, and the action logic need to be separate into provider.

The syntax for a **lightweight resource** is as follows:

require 'required\_item'

actions :action\_name1, :action\_name2, :action\_name...

default\_action :action\_name1

attribute :attribute\_name, :kind\_of => value, :name\_attribute => true

attribute :attribute\_name, :kind\_of => value, :validation\_parameter => value

...

attribute :attribute\_name, :kind\_of => value, :validation\_parameter => value

attr\_accessor :attribute, :attribute

where

* require lists any external entities that may be required by the lightweight resources, such as a library; a lightweight resource is Ruby and anything that can be done in Ruby can be done in a lightweight resource
* :action\_name1, :action\_name2, and :action\_name... represents a comma-delimited list of actions that are available to this lightweight resource; there must be at least one action
* action\_name1 is set to be the default\_action
* :attribute\_name is the name of the attribute; a lightweight resource may define as many attributes as necessary
* :kind\_of => value specifies the Ruby class (or an array of Ruby classes) that are used to define this attribute’s value
* :name\_attribute is associated with one attribute to indicate which attribute’s value will be defined by the name of the resource as it is defined in the recipe (i.e. the string that appears in front of the do block in the recipe and after the resource: resource\_name "name\_attribute" do)
* :validation\_parameter represents a comma-delimited list of validation parameters for each attribute
* attr\_accessor allows the lightweight resource to use the Module Ruby class to check for one (or more) named attributes, such as :exists or :running

The basic syntax for a **lightweight provider** that is built to leverage platform resources is as follows:

**def** whyrun\_supported?

true

**end**

use\_inline\_resources

action :action\_name **do**

condition test

resource "resource\_name" **do**

Chef::Log.log\_type "log\_message"

*# a Chef recipe*

new\_resource.updated\_by\_last\_action(true)

**end**

**end**

**end**

**def** test()

*# some Ruby code*

**end**

where:

* whyrun\_supported? indicates whether a lightweight provider can be run in why-run mode
* use\_inline\_resources is used to tell the chef-client to execute action blocks as part of a self-contained chef-client run. Using this method ensures that the chef-client can notify parent lightweight resources after embedded resources have finished processing
* action is the code block that tells the chef-client what to do when the :action\_name is used in a recipe
* condition is a Ruby condition statement (if, else, elseif, unless, while, until, case, or for)
* test is used to test for idempotence; test can be defined inline (within the action block), defined as a method using a definition block elsewhere in the lightweight provider (shown as def test()), or defined using any other pattern that is available in Ruby
* resource is a resource written as a recipe
* Chef::Log.log\_type is used to tell the chef-client to create a log entry, where log\_type is one of the following types:debug, info, warn, error, or fatal
* updated\_by\_last\_action is used to notify that a node was updated successfully

**BerkShelf:**

This is a very useful tool for managing the cookbook dependencies. Instead of explicitly downloading the dependent cookbooks while using the knife cookbook site option, this tool helps download and install all the dependent cookbooks, and also takes care of uploading the dependent cookbooks. You just need to create a Berksfile within the cookbook folder and mention the dependent cookbooks there , version constraint is optional.

**Installation:**

Multiple ways of installing Berkshelf:

* sudo gem install berkshelf – If this command fails, use the below 2 commands.
* sudo apt-get install ruby1.9.1-dev
* sudo apt-get install make

If you face installation issues with the above steps, use the below. It installs chef-dk which takes care of installing the Berkshelf along with some more additional tools. The ChefDK defines a common workflow for cookbook development, including unit and integration testing, identifying lint-like behavior, dedicated tooling, and more

* wget <https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64/chefdk_0.2.0-2_amd64.deb>
* sudo dpkg -i chefdk\_0.2.0-2\_amd64.deb
* chef verify – to verify that chef-dk got installed correctly.

**Information to use Berkshelf:**

Below is a sample entry in the Berksfile

source "https://supermarket.getchef.com"

cookbook 'nginx', '~> 1.7.0'

metadata

* source – let Berkshelf know what community site to use as source for all cookbooks.
* metadata – this option tells Berkshelf to read the metadata.rb file to find all the required cookbooks.

After telling Berkshelf where to find all the required cookbook names, we use it to install all those cookbooks:

**berks install**

Berkshelf stores cookbooks in ~/.berkshelf/cookbooks by default. This keeps your Chef repository clutter free. Instead of having to manage all the required cookbooks inside your own Chef repository, Berkshelf takes care of them. You simply need to check in Berksfile with your cookbook, and everyone using your cookbook can download all the required cookbooks using Berkshelf.

Berkshelf will only use the exact versions specified in the Berksfile.lock file, if it finds this file.

Finally, we use Berkshelf to upload all the required cookbooks on the Chef Server:

berks upload

Use the –force option to upload the cookbooks irrespective of any changes.

It by default uses the configuration of knife(knife.rb). If any attributes need to be overridden they need to be placed at ~/.bershelf/config.json. for e.g. the berks upload command doesn’t work

You still need to mention the dependency in the metadata.rb if you need to use the dependent cookbook recipes in your cookbook.

**Wrapper Cookbooks:**

This can be considered as a design pattern while writing new cokbooks which use the community cookbooks. Mostly used in case of creating application cookbooks. Basically, you use the community cookbooks for installing different and customize it to your need by overriding the attributes, templates,etc.

Below is the content of a recipe which shows an example of overriding a template from the dependent cookbook:

include\_recipe 'ntp::ntpdate'

resources("template[/etc/default/ntpdate]").cookbook "my-ntp"

Here, we tell Chef to use the ntpdate recipe from ntp cookbook, but override the template resource from the my-ntp cookbook.