Chef

Introduction

System administrators have always tried to automate repetitive tasks. Some use BASH and Perl with SSH to loop through a list of servers one at a time. This has some limitations and doesn't scale particularly well, especially in a medium to large datacenter or cloud deployment. So what is a forward-thinking system administrator to do? Let's take a look at a tool called Chef.

What is Chef?

Chef (http://www.opscode.com/chef) is one of a new breed of open-source "infrastructure as code" tools to manage infrastructures of any size. It provides administrators with the capability to define "cookbooks" that can be applied repeatedly and consistently to server and application configurations. Unlike some other "infrastructure as code" tools, Chef was designed from the beginning with systems integration in mind. This allows Chef to gather the information a server needs to configure itself. For instance, if you were to configure Apache httpd as a proxy for a farm of Tomcat instances, your configuration could ask the Chef server for a list of all the installations of Tomcat, their hostnames, and port number.

Common Terminology

The following are some Chef terms that will be helpful to know before we start:

Node: A managed machine. When the Chef client runs, it executes the configuration for a node.

Client: An authorized user of the Chef API. In most cases, every machine you manage will be represented by a) A client for logging into the API and b) A node configuration to apply. Administrators and the web interface are also clients.

Cookbook: A collection of attributes, recipes, custom resources, and definitions to configure a certain application or service. For instance you will find shared cookbooks available on the web for NTP, Apache httpd, MySQL, nginx, and other common services.

Recipe: A list of resources that should be applied to a node. Recipes are written in normal Ruby extended by the Chef Resource domain-specific language. This gives you the power of anything you can do in Ruby (conditionals, using gems, etc.) while not having to be verbose in managing the resources that make up your configuration

Chef Environments

There is more than one way to bring Chef to your environment. You can use a local set of cookbooks with the Chef client (Chef Solo), a central set of cookbooks on a local Chef server, or a central set of cookbooks on the Hosted Chef

Chef Solo

If you are thinking this is too complicated, check out chef-solo (http://wiki.opscode.com/display/chef/Chef+ Solo). It allows you to pass a set of cookbooks (either in a local directory or in a tarball on a server) and a JSON file to configure your system. While this is the simplest thing that could possibly work, you give up the ability to search for other nodes or look up items in data bags to configure your nodes.

Local Chef Server

Setting up your own Chef server has been streamlined by a cookbook provided by Opscode. The cookbook can set up the Chef server on Ubuntu 8.10+, Debian 6, and CentOS 5 Information about bootstrapping your own Chef server can be found at:
http://wiki.opscode.com/display/chef/Bootstrap+Chef+RubyGems+Installation
If you need to setup your Chef server on a different platform, check out:
http://wiki.opscode.com/display/chef/Manual+Chef+Server+Configuration.

Hosted Chef

Hosted Chef is a service provided by Opscode, the makers of Chef, which frees you from configuring your own Chef server. You can find out more at http://www.opscode.com/hosted-chef/.

COMPONENTS OF CHEF

When using Chef as a client-server application, either with a local Chef server or the Hosted Chef, there are a few moving parts to keep track of.

Knife: Knife is the tool you will use as a system administrator to interact with the server most often, especially taking cookbooks and other custom configurations and loading them into the server for distribution to clients. You can also bootstrap new servers with the Chef Client components, start new instances on the major cloud providers (AWS, Rackspace, SliceHost, and Terremark), and search for nodes or other data. Running knife --help will give you a list of supported commands.

Chef Client: The Chef client runs on the servers you are managing. It gathers information about itself using Ohai, synchronizes the cookbooks it needs from the Chef server, compiles the collection of resources that make up the configuration, and then "converges" the resources it has compiled with the state of the current machine.

Web-UI: Chef includes a web interface that lets you browse cookbooks and browse and edit nodes, roles, and clients.

Server/API: The Chef server sits at the center of the system. The Chef server exposes a RESTful API, which is used by the other components in the system. Your managed nodes, knife, and the web interface are all clients of the API. While you will probably not need to access the API directly, it is good to know that all of your data and more advanced automation are available using the API.

OHAI

Knowing the current state of your machine is a big part of any configuration management system. Ohai is a Ruby library that gathers information about your system and formats it as JSON for use by the Chef client and for storage on the Chef server. To see for yourself, type ohai at the command prompt.
This information is exposed in your recipes through the node object. For instance if a particular piece of a recipe should only be called when the node is on a specific network you can use the IPaddress property as follows:

```
if node[:ipaddress] =~ /^10\.10\./
  # do something
end
```

This would check to see if the IP address of the node was a part of the 10.10.0.0 network before attempting the steps inside the block.

Ohai also allows you to write custom plug-ins for your specific environment or equipment. See: http://wiki.opscode.com/display/chef/Writing+Ohai+Plugins

Run-List

Each node has a run list, which provides the recipes and roles for a node in the order they should be applied. This is a key difference from some other configuration management tools, which attempt to have you define the relationship between every resource. While both approaches have their strengths and weaknesses, the ordered run list is easiest for most people to understand.
As of 0.10, the knife command produces cleaner output for human consumption by default. You can use the knife command to get information about a node, including its run list, as follows:

```
$ knife node show s1.mydomain.com
Node Name: s1.mydomain.com
Environment: _default
FQDN: s1.mydomain.com
IP: 1.2.3.4
Run List: role[common]
Roles: common
Recipes: chef-client, users::sysadmins, sudo
Platform: ubuntu 10.10
```

As you can see, the server in question had a run list that includes one role (common) that Expands to the recipes chef-client, users::sysadmins, and sudo. If we want to add a new role to the node, we can use knife again:

$ knife node run_list add s1.mydomain.com "role[profit]"
run_list:
role[common]
role[profit]

The run list is immediately modified and will be applied to the node the next time chef-client runs.

Cookbooks

Anatomy of a Cookbook

When you run knife cookbook create [NAME], a new directory structure is created with some sane defaults for your new cookbook. Most of the directories will be addressed in later sections, but there are two files created in the top-level directory for your cookbook that are important.

metadata.rb

The metadata.rb file is converted to JSON when the cookbook is uploaded to the Chef server. It provides critical information about the name, version dependencies, and other properties of the cookbook.

```
name "myservice"
maintainer "My Name"
maintainer_email "me@sample.com"
license "Apache v2.0"
description "Installs/Configures myservice"
long_description
IO.read(File.join(File.dirname(__FILE__), '
README.rdoc'))
version "0.0.2"
depends "ntpd"
depends "java", "~> 1.1"
```

Much of the information in the metadata.rb file is for human consumption and is displayed in the Chef web interface. The main pieces that influence the chef client behavior as of 0.10 are the version and depends statements. The version field is important with the introduction of environments in 0.10. It allows you to pin an environment to a specific version of a cookbook. This way your production environment can remain on a tested version of your configuration while you are improving your development version. Versions have to be managed by hand right now. When you use knife to upload a cookbook to the Chef server it will replace any files or recipes it finds that were uploaded with the same version number. You can prevent yourself and others from overwriting a given version of a cookbook by passing knife the --freeze flag.
The "depends" property specifies what other cookbooks are required on the client for this cookbook to work. The second depends clause in the example includes a version number. This is optional but will allow you to keep major changes to your cookbooks from breaking others if you manage your version numbers well.

NOTE: Cookbook versions are supported as of Chef 0.10

README.rdoc

If you look again at the long_description property of the metadata. rb file you will see some Ruby code reading a file called README.rdoc. The README.rdoc file is the place for your documentation on how to use your cookbook and is especially important if you are sharing your cookbooks with others, either inside your company or on the Internet. While RDoc is the default for the README, Markdown is supported and will probably become the default in a future release.

File Specificity

Chef provides a way to place different versions of files and templates to specific hosts or platforms. The default/ folder under files/ and templates/ is created for you if you if you used `knife` to create your cookbook. You can override a file in this default directory with a version that is host specific, OS and version specific, or just OS specific.

The utility versus the complexity of the file specificity system is in discussion in the community at the moment. There are clearer ways to specify template and file resources (like in the resource itself), so I caution users to not make heavy use of this functionality. It is more likely to confuse those that come after you. Just remember to place your files and templates in the default/ directory.

Resources

When you start composing your configuration, you will declare resources, which are essentially Ruby objects with the code behind them to configure your system. Behind each resource are one or more providers that tell Chef how to execute the actions you require against a certain type of system. For instance, the package resource has providers for yum, apt, gems, and more.

Log

The Log resource is the simplest resource there is. All it does is print a logging message at the level you provide (defaulting to INFO).

log "This is a message from chef"

This will generate the following message amidst the rest of the output when chef-client runs.

INFO: This is a message from chef

You can specify the level you want the message to be logged at with the level parameter.

```
log "Bad stuff happened" do
   level :error
end
```

Packages

Chef includes providers for most major package management systems. These multiple providers allow the single package resource to be used on most major UNIX-based operating systems. since the default action for a package is "install", the simplest use is as follows:

package "autoconf"

This is equivalent to:

```
Package "autoconf" do
  action :install
end
```

You can even specify the version you want and the provider that should be used (such as YUM, rubygems, etc.):

```
package "cucumber" do
  version "0.9.4"
  provider Chef::Provider::Package::Rubygems
  action :install
end
```

There are also shortcut resources that force a particular package provider to be used. For instance, the Ruby gem example above is the Same as saying:

```
gem_package "cucumber" do
  version "0.9.4"
  action :install
end
```

As always, you have the full power of Ruby to Handle edge cases, such as the package name to use when installing Apache httpd.

```
package "apache2" do
  case node['platform']
  when "centos","redhat","fedora","suse"
  package_name "httpd"
  when "debian","ubuntu"
  package_name "apache2"
end
  action :install
end
```

In the example above, we use a Ruby case statement to look at the node object, which represents the system we are currently configuring. The node object includes all the data discovered by the Ohai application. Ohai sets the platform element to the name of the distribution.

Files, Directories and Templates

Configuration on Linux and *NIX systems often starts with managing files and directories. Chef provides the file, remote_file, and ookbook_file resources to manage static files and a directory resource for managing directories.

Directory

The directory resource lets us create, remove, and manage the permissions on directories. If you need a temporary directory, you can create one as follows:

```
directory "/tmp/mydirectory" do
  action :create
end
```

The owner and group of the directory will be the defaults for the user running the Chef client, usually root. The create action is the default, so that could be rewritten as:

```
directory "/tmp/mydirectory
```

In defining your resources, you have the full power of Ruby available to you. For instance, this allows you to loop through entries in an array to build a set of directories.

```
["site1", "site2", "site3"].each do |dir|
  directory "/srv/vhosts/#{dir}" do
    mode 0775
    owner "root"
    group "root"
    action :create
    recursive true
  end
end
```

In the example above, three directories will be created under /srv/vhosts. In this case we used the recursive true property of the directories to ensure that the base directories (/srv and /srv/vhosts) are created if they do not exist.

File

The file resource allows you to manage the permissions and ownership of files on the node, with the option to pass content from inside your recipe. To retrieve a file from a URL or the cookbook, use the remote_file or cookbook_file resources respectively.
The parameters for a file are the same as a directory with a couple of exceptions. Files have a touch action that updates the modified time and last accessed time of the file, they don't have a recursive attribute so the target directory must exist, and they have a backup attribute that defines how many backup versions of the file should be kept if the contents change.

NOTE: Backups are saved to the directory specifed by file_backup_path in your client.rb file. This defaults to /var/chef/backup.

```
File "/etc/nologin" do
      backups false
      owner "root"
      group "root"
      mode "0755"
      action :create
 end
```

The resource above creates a file called /etc/banner with the content set to the string provided. It will not back up the file if it changes.

Remote File

The remote_file resource is identical to the file resource, except instead of having a content parameter, remote_file has a source parameter that is the URL of the file to transfer. It also has an optional checksum parameter that, if the managed file's checksum matches, will prevent Chef from downloading the file an extra time. Chef uses SHA-256 for its checksums, but only the first 5-10 characters of the hash are usually needed to ensure consistency.
Also, the actions on a remote_file are limited to create and create_if_missing. To delete a file created by remote_file, change it to a standard file resource.

Cookbook File

A cookbook_file resource is largely the same as a remote_file resource, with the exception that files will be retrieved from the files/ directory structure (respecting File Specificity) of the cookbook. An additional cookbook attribute is available to fetch files from cookbooks other than the one you are currently running.

Templates

Chef supports templating text based configuration files using ERB. In its simplest form, Ruby code is wrapped in special brackets.

```
<% x = "This is Ruby code" %>
```

Things that are not wrapped in the tags are not parsed as Ruby code. A similar tag is used to put a value into the resulting file.

```
The value of x is: <%= x %>
```

Notice the equals sign after the opening tag. ERB will parse the statement included in that tag and replace the tag with the return value. By combining these elements with regular Ruby conditionals and flow control, we can create templates for even complex configuration. Let us stick with the simple case for now, a resolv.conf file used by *NIX systems to define their DNS servers.

```
domain <%= node['domain'] %>
search <%= node['domain'] %>
<% @nameservers.each do |server_ip|
-%>
nameserver <%= server_ip %>
<% end -%>
```

You can see that the domain and search lines are adding the value returned by node['domain'] to the end of their lines. This shows the first way Chef makes templating a little easier. The node object with its extensive hashmap of attributes and Ohai values is injected into your template's namespace.

The second piece is a loop over each item in an array called **@nameservers**. The template resource has a variables attribute that you can pass other data to for use in your templates without inserting it into the node object itself.

You may have noticed that the each do and end statements have an extra hyphen attached to the end tag. This prevents the newline character outside the tag from being printed, keeping you from having extra blank lines in your template.

Let's take a quick look at how to declare the template resource for the file above. Remember that templates respect File Specificity, so we will need to put it in template/default/resolv.conf. erb.

```
template "/etc/resolv.conf" do
source "resolv.conf.erb"
mode "0644"
variables(:nameservers => ['8.8.8.8', '8.8.4.4'])
end
```

We set the source and mode, just like a cookbook_file resource. Then we add a variables attribute that assigns an array to the symbol nameservers. This array will be made available, as you saw in our template, as the variable @nameservers.

For more on ERB, see the documentation at http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc./

Services

Once the package is installed and you have put your configurations (either as files or templates) in place, it is time to manage the services you intend to provide. Again, Chef Providers are available for most of the major service management schemes.

```
service "ntpd" do
  action [:start, :enable]
end
```

This will start the service (if it is not running) and make sure it will restart when the node is rebooted. If it appears that Chef is attempting to start your service every time it runs, check the resources wiki page for the `supports`, `pattern`, and `status_command` attributes.

Execute

So far we can manage files, install packages, and restart services. But sometimes you just need to run a real command. Perhaps it is an installer for your monitoring system or to add records to an LDAP server.

```
execute "bundle install --deployment" do
  cwd "/srv/app"
  environment ({'HOME' =>
  '/home/myhome'})
  user "appuser"
  group "appgroup"
end
```

In the example above, the Chef client will switch to the user and group provided, add HOME to the environment variables, and change directories to "/srv/app" before running the command provided as the name of the resource.

Script (bash, csh, perl, python and ruby)

The script resource is an extension of the execute resource, supporting all of parameters of the latter, while allowing you to pass a block of commands inside the resource. This works well for running a custom compile of an application that you do not have a package for or for running a set of tasks every time the Chef client runs.

```
script "git update and garbage collection" do
    interpreter "bash"
     user "root"
     cwd "/srv/app"
     code <<-EOH
     git pull stage
     git gc
    EOH
    end
```

You can see the added code block with the Ruby specific <<- annotation to create a string out of everything it sees until the pattern provided (EOH in this case) is reached.

WRITING A COOKBOOK

Attributes

Other than Ohai, we really haven't talked about variables and how you provide different values to different machines depending on location, purpose or other metadata.

Attributes extend the node model just like Ohai does but are set or changed inside of cookbooks and roles. Cookbook attributes will generally go in the **attributes/default.rb** file. Once again, the attributes file is just Ruby with some shortcuts built in. You can use Ohai information to make decisions about the values you want to assign to your attributes.

Default, Normal and Override

Chef allows us to set defaults and provide overrides for attributes at the cookbook, environment, role, and node level. Of course if there is more than one value for an attribute we need to know which one wins. That brings us to…

Precedence

Attribute precedence can get tricky. Not only do override attributes take precedence over normal attributes (which beat out default attributes), but where an attribute is declared matters as well. As you explore more complicated infrastructure, you will need to become comfortable with attribute precedence. Instead of trying to explain it here, check out http://wiki.opscode.com/display/chef/Setting+Attributes+%28Examples%29 and try some experimentation.

Conditional Execution

There will be many cases where you want a resource to execute its action only if certain conditions are met. While some resources cover the simple cases of this (the execute resource has a property called creates that prevents the command from running if a given file already exists), you will come up with other cases in modeling your configuration that aren't covered.

Chef provides two conditional execution attributes on every resource: not_if and only_if. Both will take either a string, which will be executed as a shell command, or a Ruby block. A return code of 0 from the shell command is treated like a value of true from the Ruby block. For instance, you might only want a particular script to run if the /etc/hosts file contains the right machine name.

```
execute "run the script" do
command "runme --now"
only_if "grep myserver /etc/hosts"
end
```

An experienced BASH scripter knows that the grep command returns 0 if it fnds what you were looking for. The resource above will execute the command runme --now only when the /etc/hosts file contains the string "myserver". Passing a Ruby block is similar but gives you the full power of the Ruby language as part of your conditional.

```
execute "run the script again" do
command "runme --now"
not_if { File.exists?("/etc/passwd") }
end
```

In this case, we are trying to run the same script, but we aren't going to do it if the file /etc/passwd exists. Since we are using Ruby we can call the exists? static method on the File class.

Notifications

Sometimes you want an action to be applied to a resource if something changes. For example, if your httpd.conf file changed you would want to restart Apache. All resources support the "notifies" attribute that takes an action, a service, and optional timing.

```
template "/etc/httpd/conf/httpd.conf" do
  source "httpd.conf.erb"
  notifes :restart, "service[apache2]", :delayed
end
```

In the example above we declare a template resource that, when it changes, will tell the apache2 service to restart. The last parameter tells Chef to delay the restart to the end of the client run, which keeps you from restarting a

service over and over if multiple configuration files change during one client run.

Putting it together

So let's string together the things we have covered and create a simple cookbook for managing the NTP daemon on our servers. First lets create a couple of attributes, specifically an array of servers to poll for time changes and if we want to enable statistics on our ntpd server.

attributes/default.db
default['ntpd']['servers'] = ['tick.ucla.edu', 'tick.uh.edu']
default['ntpd']['stats_enabled'] = false

By defining these as default attributes, they can be overridden later with servers that are closer, based on role or environment. Now let's create a recipe with three resources in it: a package, a template, and a service.

```
recipes/default.rb
  package "ntpd" do
    action :install
  end
  template "/etc/ntpd.conf" do
    source "ntpd.conf.erb"
    owner "0755"
    notifes :restart, "service[ntpd]"
  end
  service "ntpd" do
    action [:start, :enable]
  end
```

The package resource installs the NTP, while the template and service make sure it is configured and running. When the template is updated it notifies the service to restart.

The only thing we need now is our NTP configuration template (snipped for length).

```
templates/default/ntpd.conf.erb
driftfile /var/lib/ntp/ntp.drift
<% if node['ntpd']['stats_enabled'] -%>
statsdir /var/log/ntpstats/
<% end -%>
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day
enable
<% node['ntpd']['servers'].each do |srv| -%>
server <%= srv %>
<% end -%>
--- SNIP ---
```

ROLES

The run list of a node either contains recipes or roles. Roles have their own run list, allowing you to group common functions or service types together and not repeating yourself with a long run list on each of your application servers.

```
name "webservers"
description "HTTP server role"
run_list "recipe[apache2]", "recipe[ntpd]"
default_attributes "ntpd" => { "stats_enabled"
=> true }
--- SNIP ---
```

This role applies the apache2 and ntpd recipes and changes the value of the stats_enabled attribute to true. Roles can contain other roles as well as recipes in their run list, which allows you to build a hierarchy of roles to model your environment. Most environments start with a base or common role that holds the pieces that apply to every node and then grow to include individual application.

Chef (Further Reading)

So now you have an idea how to build your cookbooks. There are other pieces to make your life easier and your automation cleaner. Since I cannot cover them all, here is a quick list so you know what you can look for.

Chef Data Bags:
https://docs.chef.io/data_bags.html

Chef Search:
https://docs.chef.io/chef_search.html

Environment:
https://docs.chef.io/environments.html

Chef Automate
https://www.chef.io/automate