**Chef's Two Pass Model**2015-10-09

A common source of bugs in Chef code for new and experienced users alike is Chef’s two-pass (or two-phase) execution model. I would like to provide a quick overview of this system and how it can impact your code.

**Phases**

Chef’s loading process can be broadly split into four phases:

1. Load
2. Compile
3. Converge
4. Cleanup

Of these, the compile and converge phases make up the bulk of the work and are “execution phases”. When we talk about the two-pass or two-phase system, it is these two phases we are talking about, but let’s look at each phase in turn.

**Load**

During the load phase Chef syncs all the needed cookbooks with the Chef Server if one is being used. Each of the five types of support files are loaded in order:

1. libraries/
2. attributes/
3. resources/
4. providers/
5. definitions/

All of the files of each type are loaded for all cookbooks before moving on to the next type. Loading happens in sorted (topographic) order with respect to cookbook dependencies and the run list order, and alphabetical order within each cookbook.

This means if cookbook B depends on cookbook A, we might see a load order like:

1. a/libraries/default.rb (library files are first, A is sorted before B because of dependency)
2. b/libraries/default.rb (then B’s library files)
3. a/attributes/default.rb (attribute files are next, again A is first)
4. b/attributes/default.rb (and B is second because B depends on A)
5. b/resources/first.rb (A has no resources files so we skip to B, first is alphabetically before second)
6. b/resources/second.rb

By the time the loading phase is complete, the node object is fully populated, all custom resources are available for use, and we can move on to the compile phase.

**Compile**

The compile phase is first of the execution phases. The goal of the compile phase is to go from recipe source code to in-memory representations of resource objects. At this point we have taken the node’s run list and fully expanded it, so any roles are replaced by their constituent recipes until all we have is an ordered list of recipes to run. We will run each of those recipes in order, and if no errors are raised we’ll move on to the next phase.

This is where the major “gotcha” of the execution model comes in. Take some example recipe code like:

file *'/foo'* **do**

content *'bar'*

**end**

Assuming that recipe is in the run list, or is run via include\_recipe from something else being compiled, that code will run. However running the code does not actually write any content to file. All it does is create aChef::Resource::File object with all of the data expressed in the recipe code and add that object to the resource collection. This is true of all resources in the DSL, running (compiling) the recipe file is just queuing up resources in the resource collection. No changes to the system should happen until the converge phase. This also means that any Ruby code in the file not explicitly delayed (ruby\_block, lazy, not\_if/only\_if) is run when the file is run, during the compile phase.

**Converge**

Once the compile phase completes we have a resource collection that is fully loaded and ready to go. This is an array of resource objects that represent the data from our recipes. The majority of the converge phase can be seen as:

resource\_collection.each **do** |resource|

resource.run\_action(resource.action)

**end**

There is a little more complexity to deal with things like notifications, but that is the heart of it; loop over each resource and run the requested action. This is where provider classes get used, run\_action creates a provider instance internally and runs the action code from the provider. Those methods are what actually do all the interesting things like writing files, installing packages, etc.

**Cleanup**

With the compile phase finished, we just have a few cleanup steps left to process. This includes things like running handler plugins, saving the node state back up to the Chef Server, and sending data to the Chef Analytics server if being used.

**Bad Code**

A concrete example of the kind of errors that can creep in due to the execution model:

file *'/foo'* **do**

content *'bar'*

**end**

**if** File.exist?(*'/foo'*)

execute *'myapp /foo'*

**end**

Here we have two resources and an if statement. The bug happens because the File.exist? check will run at compile time, and even though it is after the file resource, at that point in the execution the file hasn’t actually been written yet. The solution in this case is to use an only\_if guard like:

file *'/foo'* **do**

content *'bar'*

**end**

execute *'myapp /foo'* **do**

only\_if { File.exist?(*'/foo'*) }

**end**

As only\_if guard clauses (and lazy property values and the block onruby\_block) are run during the converge phase, this will more often behave as the author expects.

**[tl;dr](https://coderanger.net/two-pass/" \l "tldr)**

First all recipes on the run list are compiled in to resources in the resource collection, then all resources in the collection are converged.

Any code outside lazy, only\_if/not\_if, or a ruby\_block’s block property is run at compile time, which is before any resource runs its actions.

Infrastructure automation is the process of scripting environments — from installing an operating system, to installing and configuring servers on instances, to configuring how the instances and software communicate with one another, and much more. By scripting environments, you can apply the same configuration to a single node or to thousands.

Infrastructure automation also goes by other names: configuration management, IT management, provisioning, scripted infrastructures, system configuration management, and many other overlapping terms. The point is the same: you are describing your infrastructure and its configuration as a script or set of scripts so that environments can be replicated in a much less error-prone manner. Infrastructure automation brings agility to both development and operations because any authorized team member can modify the scripts while applying good development practices — such as automated testing and versioning — to your infrastructure.

##### About this series

Developers can learn a lot from operations, and operations can learn a lot from developers. This series of articles is dedicated to exploring the practical uses of applying an operations mindset to development, and vice versa — and of considering software products as holistic entities that can be delivered with more agility and frequency than ever before.

In the past decade, several open source and commercial tools have emerged to support infrastructure automation. The open source tools include Bcfg2, CFEngine, Chef, and Puppet. They can be used in the cloud and in virtual and physical environments. In this article, I'll focus on the most popular open source infrastructure automation tools: Chef and Puppet. Although you won't learn the intricacies of either tool, you'll get an understanding of the similarities and differences between them, along with some representative examples. For a more detailed example of setting up and using an infrastructure automation tool, this article provides a [companion video](http://www.youtube.com/watch?feature=player_embedded&v=-JXZLCmRQF4) that shows how to run Puppet in a cloud environment.

##### Traditional approaches

Not all teams are applying infrastructure automation tools — along with its practices and patterns — so what are they doing? Traditional approaches — which do not scale — include configuring environments manually or writing and running combinations of scripts that must be performed by a human. This leads to error-prone processes that increase cycle times, preventing teams from regularly releasing software.

Chef and Puppet both use a Ruby domain-specific language (DSL) for scripting environments. Chef is expressed as an internal Ruby DSL, and Puppet users primarily use its external DSL — also written in Ruby. These tools tend to be used more often in Linux® system automation, but they have support for Windows as well. Puppet has a larger user base than Chef, and it offers more support for older, outdated operating systems. With Puppet, you can set dependencies on other tasks. Both tools are idempotent— meaning you get the same result with the same configuration no matter how many times you run it.

## Chef

Chef has been around since 2009. It was influenced by Puppet and CFEngine. Chef supports multiple platforms including Ubuntu, Debian, RHEL/CentOS, Fedora, Mac OS X, Windows 7, and Windows Server. It is often described as easier to use — particularly for Ruby developers, because everything in Chef is defined as a Ruby script and follows a model that developers are used to working in. Chef has a passionate user base, and the Chef community is rapidly growing while developing cookbooks for others to use.

### **How it works**

##### Get involved

developerWorks [Agile transformation](https://www.ibm.com/developerworks/agile/) provides news, discussions, and training to help you and your organization build a foundation on agile development principles.

In Chef, three core components interact with one another — Chef server, nodes, and Chef workstation. Chef runs cookbooks, which consist of recipes that perform automated steps — called actions — on nodes, such as installing and configuring software or adding files. The Chef server contains configuration data for managing multiple nodes. The configuration files and resources stored on the Chef server are pulled down by nodes when requested. Examples of resources include file, package, cron, and execute.

Users interact with the Chef server using Chef's command-line interface, called Knife. Nodes can have one or more roles. A role defines attributes (node-specific settings) and recipes for a node and can apply them across multiple nodes. Recipes can run other recipes. The recipes in a node, called a run list, are executed in the order they are listed. A Chef workstation is an instance with a local Chef repository and Knife installed on it.

Table 1 describes the core components of Chef:

##### **Table 1. Chef components**

| **Component** | **Description** |
| --- | --- |
| Attributes | Describe node data, such as the IP address and hostname. |
| Chef client | Does work on behalf of a node. A single Chef client can run recipes for multiple nodes. |
| Chef Solo | Allows you to run Chef cookbooks in the absence of a Chef server. |
| Cookbooks | Contain all the resources you need to automate your infrastructure and can be shared with other Chef users. Cookbooks typically consist of multiple recipes. |
| Data bags | Contain globally available data used by nodes and roles. |
| Knife | Used by system administrators to upload configuration changes to the Chef Server. Knife is used for communication between nodes via SSH. |
| Management console | Chef server's web interface for managing nodes, roles, cookbooks, data bags, and API clients. |
| Node | Hosts that run the Chef client. The primary features of a node, from Chef's point of view, are its attributes and its run list. Nodes are the component to which recipes and roles are applied. |
| Ohai | Detects data about your operating system. It can be used stand-alone, but its primary purpose is to provide node data to Chef. |
| Recipe | The fundamental configuration in Chef. Recipes encapsulate collections of resources that are executed in the order defined to configure the nodes. |
| Repository (Chef repository) | The place where cookbooks, roles, configuration files, and other artifacts for managing systems with Chef are hosted. |
| Resource | A cross-platform abstraction of something you're configuring on a node. For example, users and packages can be configured differently on different OS platforms; Chef abstracts the complexity in doing this away from the user. |
| Role | A mechanism for grouping similar features of similar nodes. |
| Server (Chef server) | Centralized repository of your server's configuration. |

### **Examples**

Listing 1 demonstrates the use of the service resource within a recipe that's part of a Tomcat cookbook. You can see that you can use tools like Chef to do platform-specific configuration and manage server configuration.

##### **Listing 1. Chef recipes**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | service "tomcat" do    service\_name "tomcat6"    case node["platform"]    when "centos","redhat","fedora"      supports :restart => true, :status => true    when "debian","ubuntu"      supports :restart => true, :reload => true, :status => true    end    action [:enable, :start]  end |

Listing 2 defines the attributes for the Tomcat cookbook. In this example, I'm defining some external ports for the Tomcat server to make available. Other types of attributes you might see include values for directories, options, users, and other configurations.

##### **Listing 2. Chef attributes**

|  |  |
| --- | --- |
| 1  2  3 | default["tomcat"]["port"] = 8080  default["tomcat"]["ssl\_port"] = 8443  default["tomcat"]["ajp\_port"] = 8009 |

Chef extends the Ruby language — as compared to an external DSL — to provide a model for applying configuration to many nodes at once. Chef uses an imperative model without explicit dependency management, so people with more of a development background tend to gravitate toward Chef when they are scripting environments.