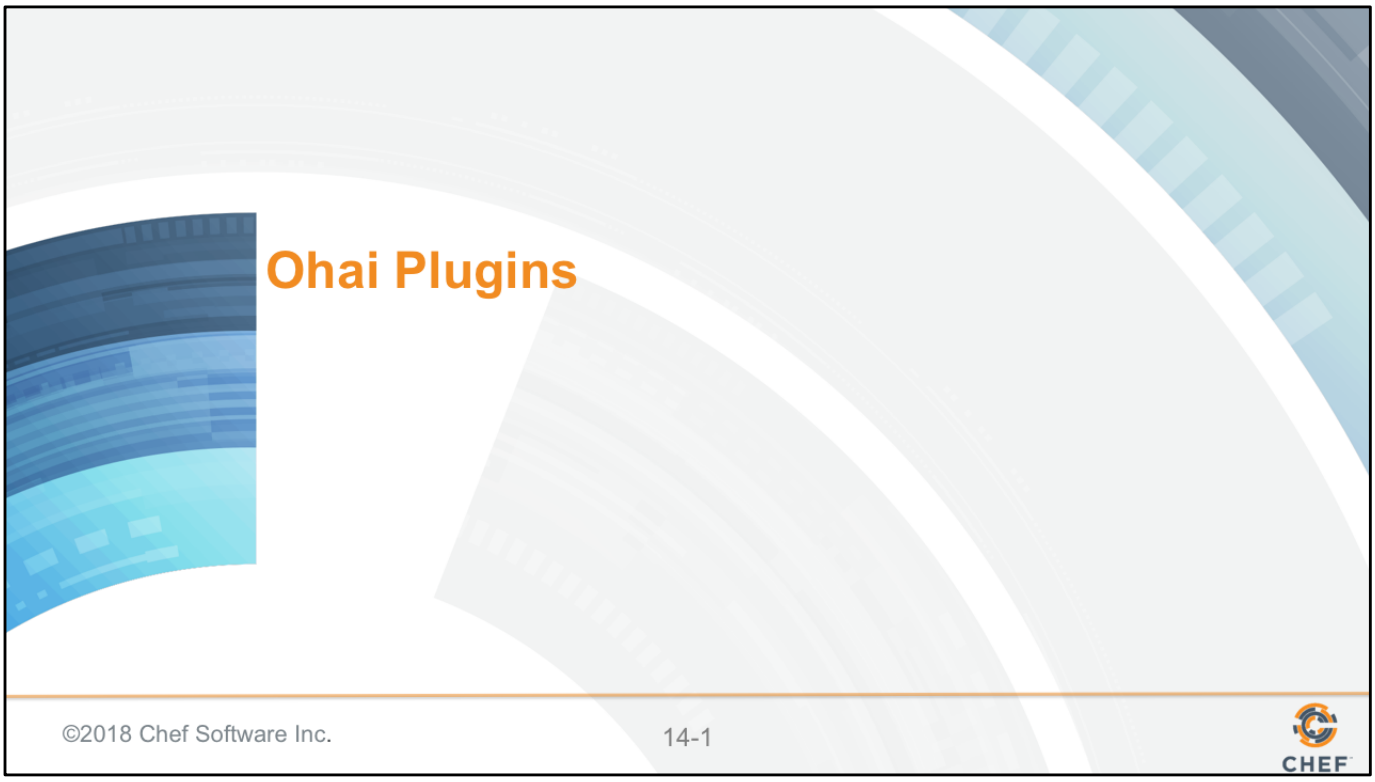
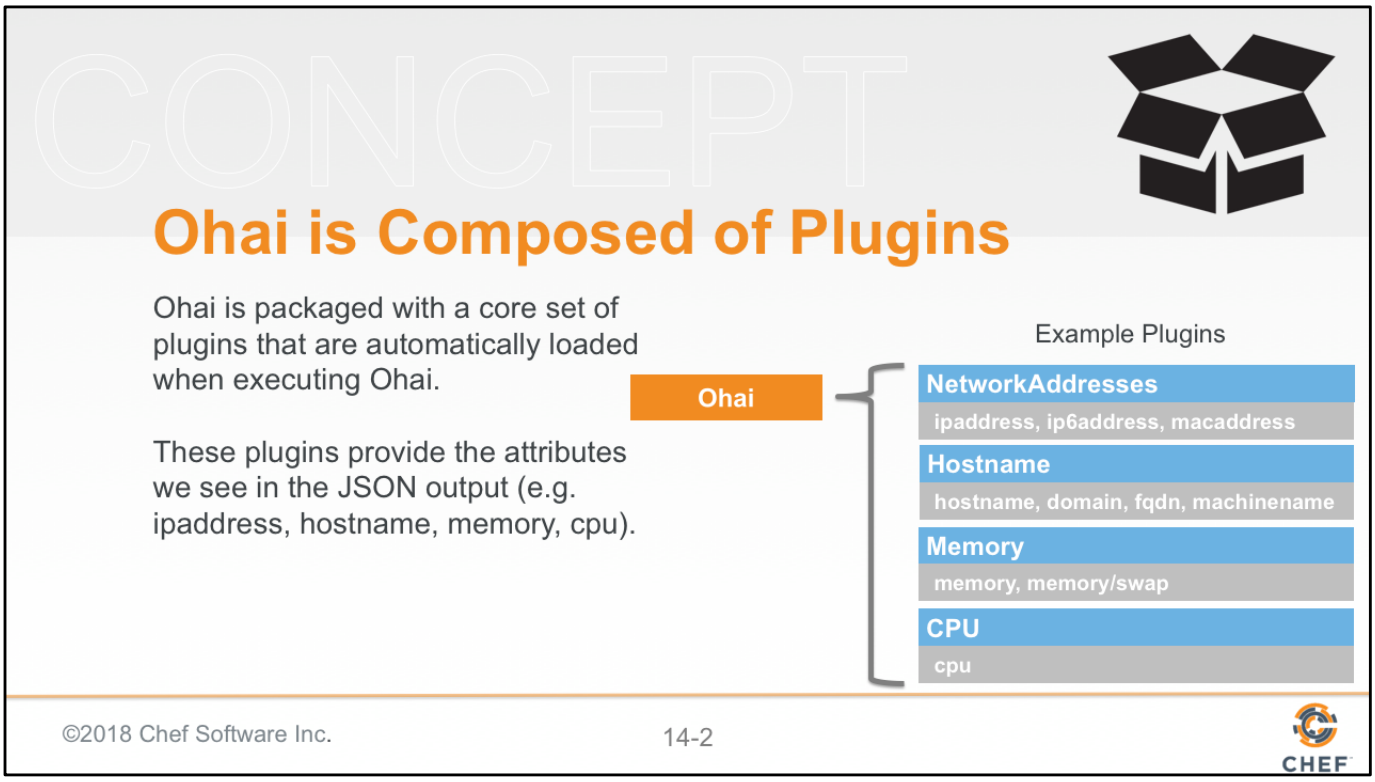

Ohai Plugins
©2018 Chef Software Inc.
14-1
CHEF



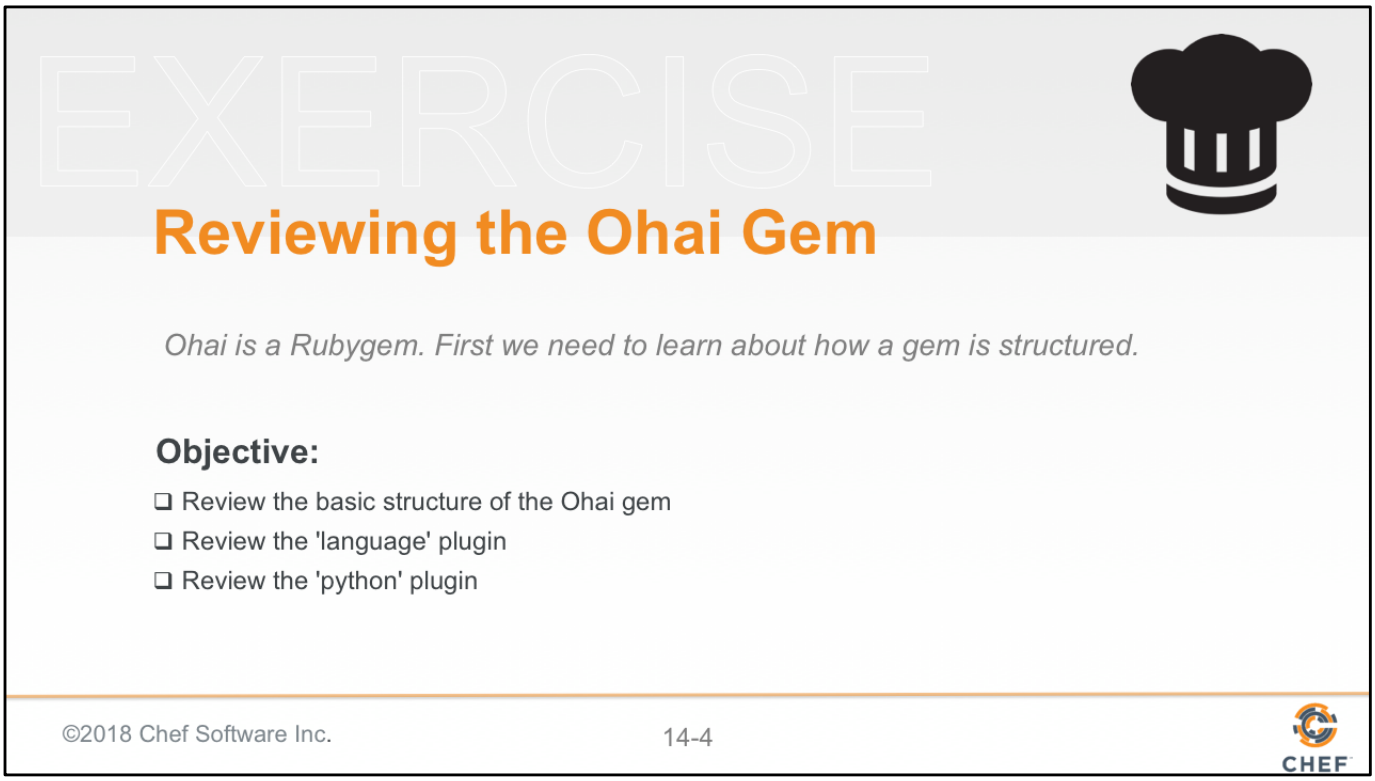
As we saw in the previous module Ohai provides a large set of attributes that it provides through plugins. All the data that Ohai collects are stored in plugins. Ohai comes packaged with a core set of plugins that capture a lot of common data across many different platforms.

Objectives

After completing this module, you should be able to:

- Find Ohai's core plugins
- Express what a plugin provides, depends on, and how it collects its data

After completing this module you will be able to find the plugins that come packaged core with Chef, express what a plugin provides, depends on, and how it collects its data



To review the core plugins packaged with Ohai we need to spend some time reviewing the source code of the gem as none of the gems are not defined in documentation.

CONCEPT

Ohai is Ruby Gem

Ruby gems are the ways in which Ruby developers share the code that they develop with others. A Ruby gem is really a packaging structure similar to that of a Chef cookbook.

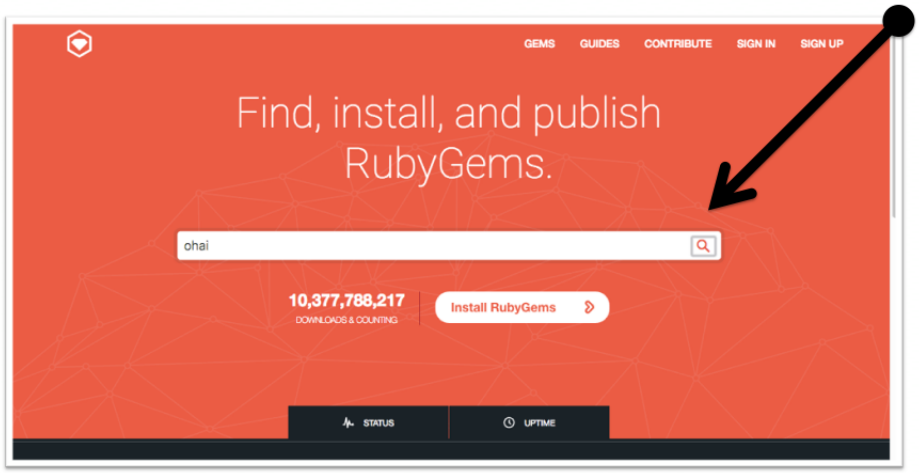
CHEF

Ohai is a Ruby gem that is packed in the Chef Development Kit (Chef DK). A Ruby gem is a packaging structure that allows for the code to be reused and shared.

Searching for a Gem on Rubygems

Steps

1. Visit https://rubygems.org
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.



CHEF

All Rubygems are stored on rubygems.org. We can come to the site and search for any gem by their name. Search for the Rubygem named "ohai".

Searching for a Gem on Rubygems

Steps

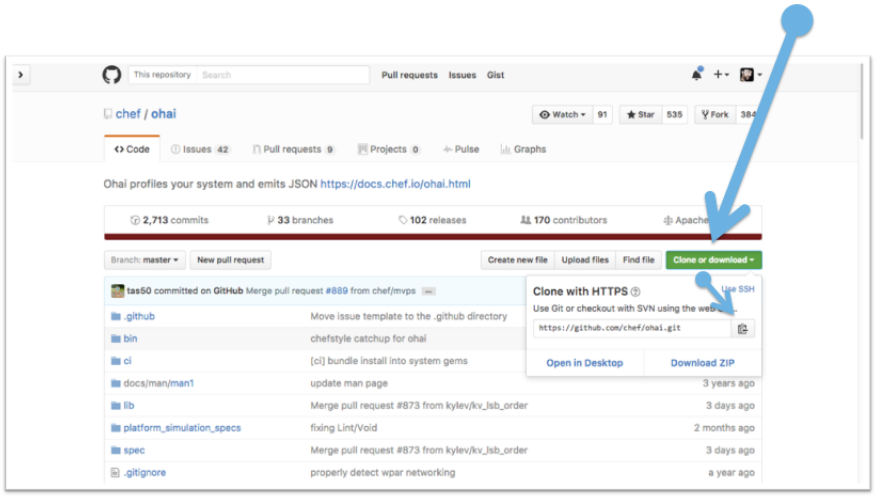
1. Visit https://rubygems.org
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.

The project page for the gem itself contains important information about the releases, where to find the source, where to file issues, etc. We are interested in viewing the source of the project so we want to click on that link.

Searching for a Gem on Rubygems

Steps

1. Visit https://rubygems.org
2. Within the search field enter: **ohai**
3. Press enter or click the magnified glass at the right-side of the search box.
4. Click the 'Source Code' link
5. Click on 'Clone or download' and then copy the git URL.



The ohai project is stored as a git repository within the Chef organization on GitHub. We can clone this project to our workstation to give us the ability to review the source code.

We are going to obtain the Ohai library on our local workstation so let's start by returning to the home directory.

Cloning the Ohai Library

```
> git clone https://github.com/chef/ohai.git
```

```
Cloning into 'ohai'...
remote: Counting objects: 20571, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 20571 (delta 7), reused 0 (delta 0), pack-reused 20540
Receiving objects: 100% (20571/20571), 4.46 MiB | 2.13 MiB/s, done.
Resolving deltas: 100% (13408/13408), done.
Checking connectivity... done.
```


CHEF

Git is installed with the Chef DK so we will use it to clone the Ohai project.

Viewing the Contents of the Project

```
> tree ohai
```

```
└── ohai
    ├── appveyor.yml
    ├── bin
    │   └── ohai
    ├── CHANGELOG.md
    ├── ci
    │   ├── jenkins_run_tests.bat
    │   └── jenkins_run_tests.sh
    ├── docs
    │   └── man
    │       └── man1
    │           └── ohai.1
```


The gem contains several important items within the top-level directory. We are going to explore the contents of some of the essential files.

Viewing the README

~/ohai/README.md

```
# ohai

[![Build Status Master](https://travis-ci.org/chef/ohai.svg?branch=master)](https://travis-
ci.org/chef/ohai) [![Build Status
Master](https://ci.appveyor.com/api/projects/status/github/chef/ohai?branch=master&svg=true&p
assingText=master%20-%20Ok&pendingText=master%20-%20Pending&failingText=master%20-
%20Failing)](https://ci.appveyor.com/project/Chef/ohai/branch/master) [![Gem
Version](https://badge.fury.io/rb/ohai.svg)](https://badge.fury.io/rb/ohai)


## Description

Ohai detects data about your operating system. It can be used standalone, but its primary
purpose is to provide node data to Chef.

Ohai will print out a JSON data blob for all the known data about your system. When used with
Chef, that data is reported back via node attributes.

Chef distributes ohai as a RubyGem. This README is for developers who want to modify the Ohai
source code. For users who want to write plugins for Ohai, see the docs:
```


The README contains information on how to install, configure and use this gem. This is often the place to start when exploring the gem.

Viewing the Gem Specification

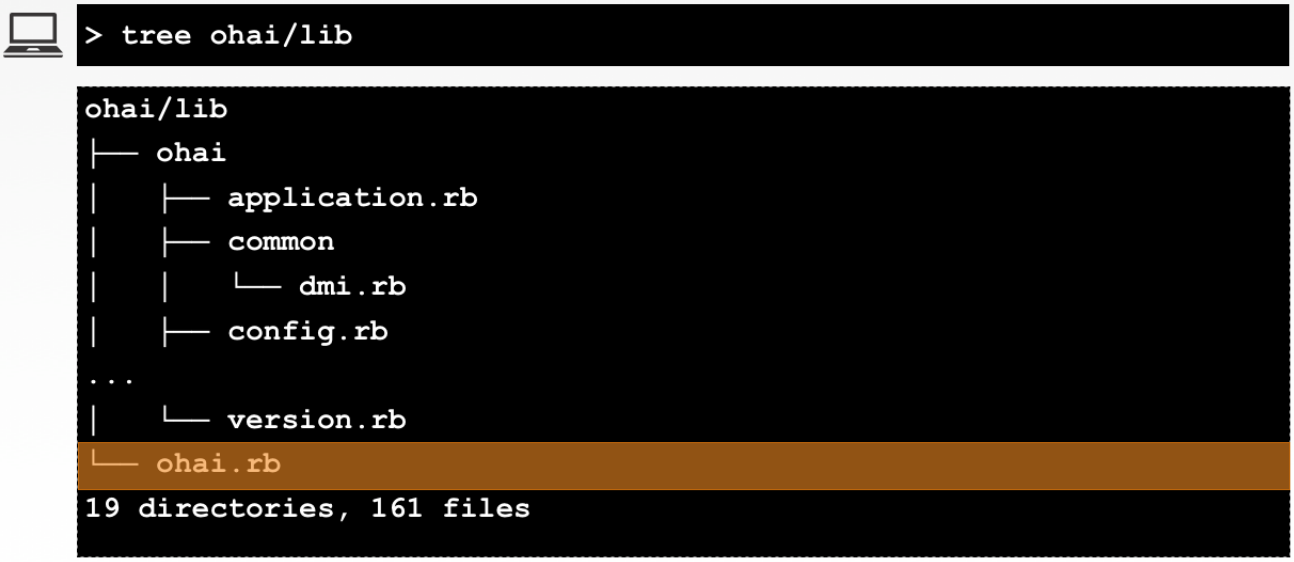
~/ohai/ohai.gemspec

```
$:.unshift File.expand_path("../lib", __FILE__)
require "ohai/version"

Gem::Specification.new do |s|
  s.name = "ohai"
  s.version = Ohai::VERSION
  s.platform = Gem::Platform::RUBY
  s.summary = "Ohai profiles your system and emits JSON"
  s.description = s.summary
  s.license = "Apache-2.0"
  s.author = "Adam Jacob"
  s.email = "adam@chef.io"
  s.homepage = "https://docs.chef.io/ohai.html"
```


The gem specification defines important information about the Rubygem. Within it you will find metadata that describes the owner, licensing, contact information, dependencies, development dependencies, the files to package in the gem, and which one of those are executables.

Viewing the lib Directory



The lib (or library) directory contains the source code for this gem. Within the root of the directory you will find a single file that shares the same name as the gem.

In the previous module when we typed "require 'ohai'" this was the file that was loaded into memory.

Viewing the ohai.rb file in the lib Directory

~/ohai/lib/ohai.rb

```
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

require "ohai/version"
require "ohai/config"
require "ohai/system"
require "ohai/exception"
```


This file requires more files from within the gem. The paths specified are relative to the 'lib' directory so all of these examples are loading files from within the subdirectory of ohai.

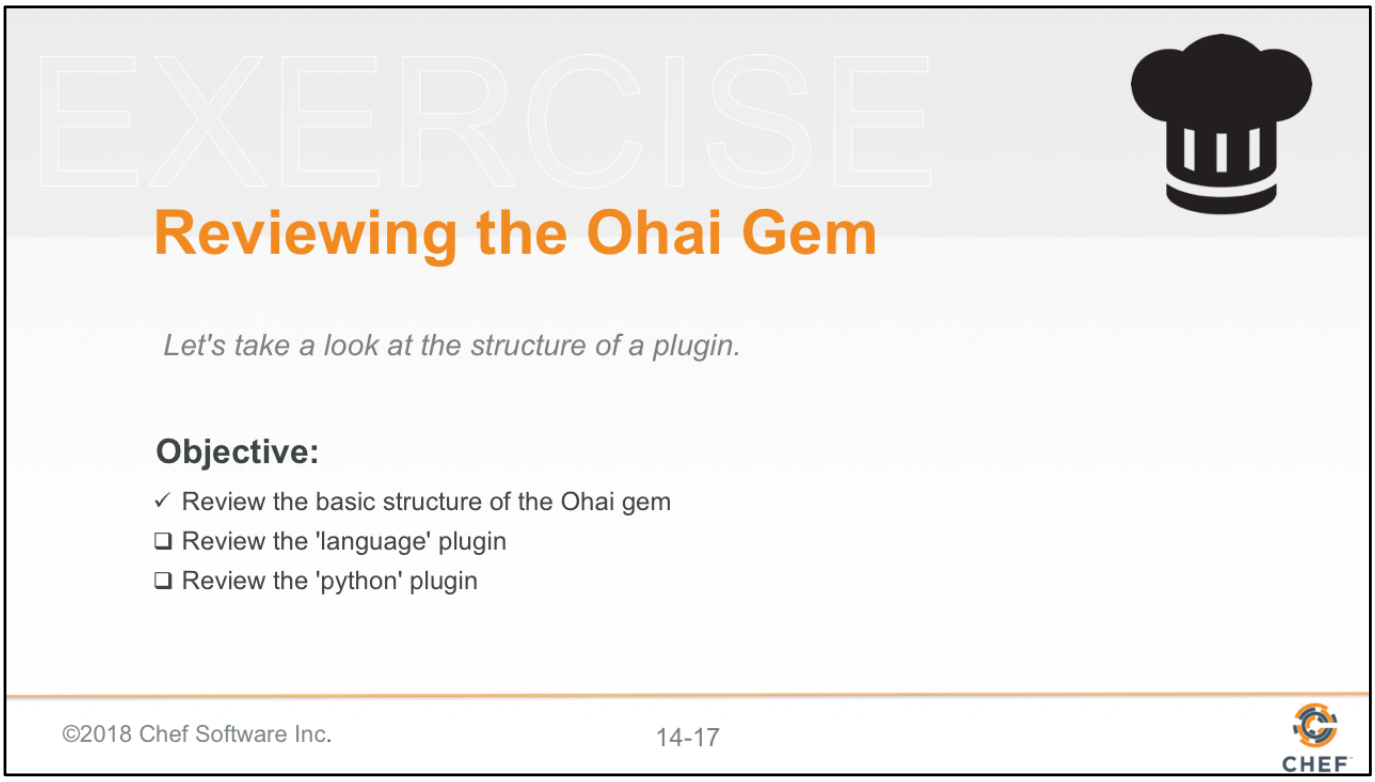
Viewing the plugins directory

```
> tree ohai/lib/ohai/plugins
```

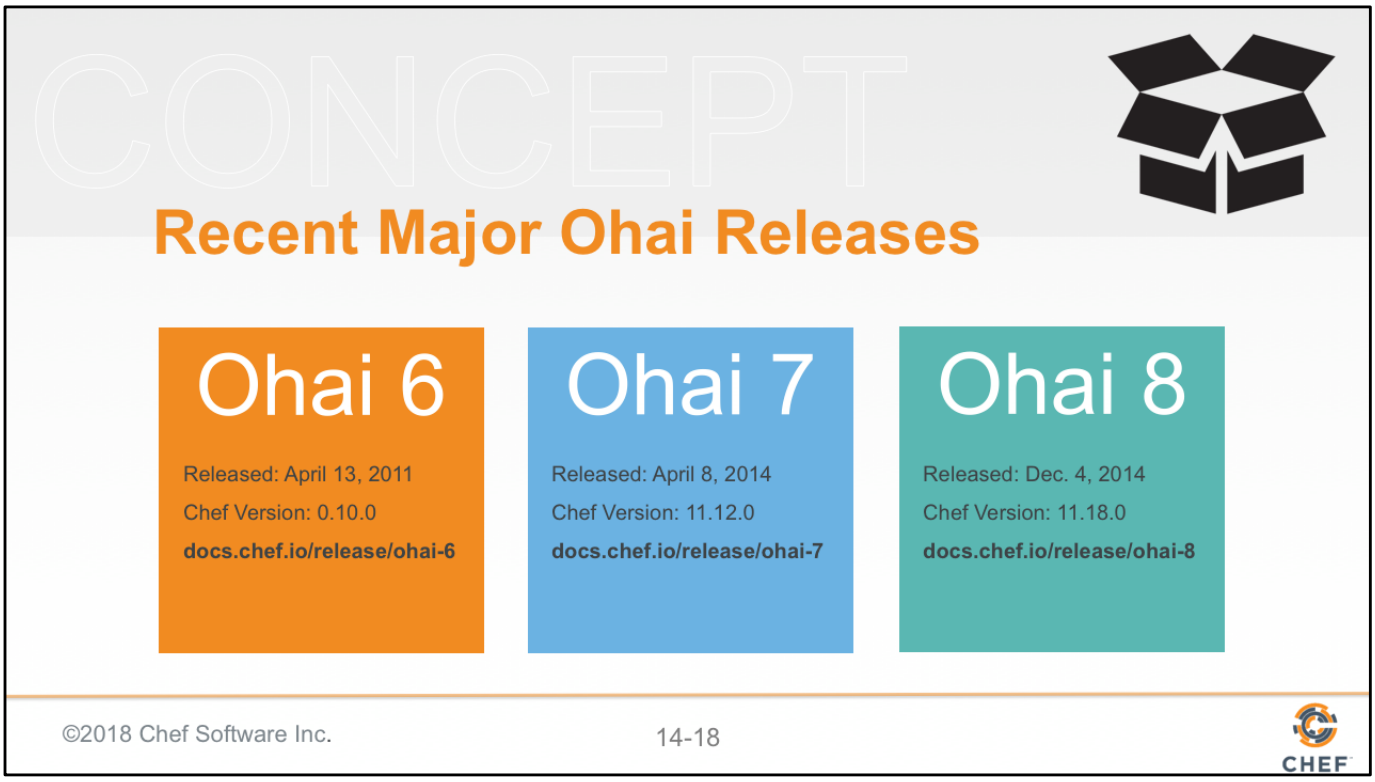
```
ohai/lib/ohai/plugins
├── aix
│   ├── cpu.rb
│   ├── filesystem.rb
│   ├── kernel.rb
│   ├── memory.rb
│   ├── network.rb
│   ├── os.rb
│   ├── platform.rb
│   ├── uptime.rb
│   └── virtualization.rb
├── azure.rb
├── bsd
│   ├── filesystem.rb
```


CHEF

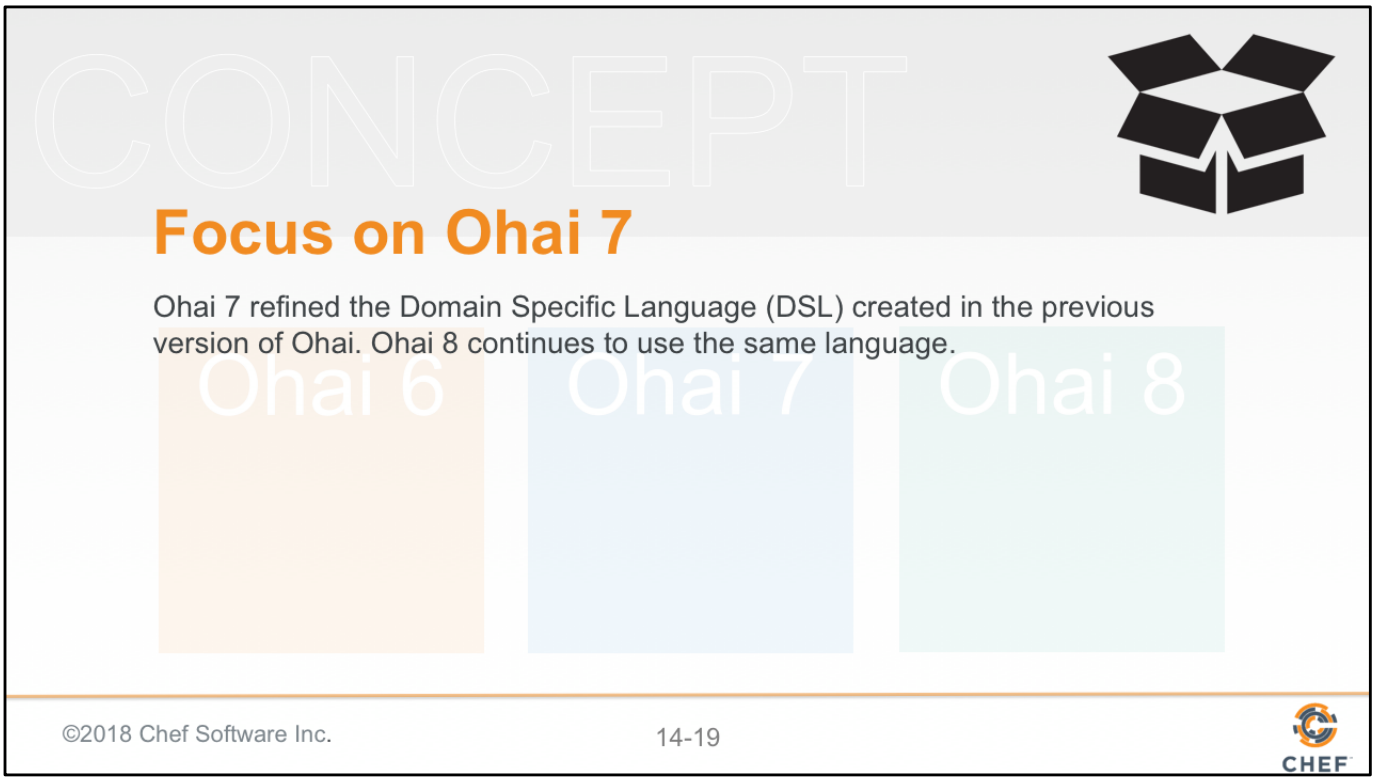
Ohai stores its plugins in a specific subdirectory of this project.



That was a quick introduction to the gem structure to give us an idea about where the plugins are stored. Now it is time to explore the Domain Specific Language (DSL) used to write these plugins.



Ohai has seen many notable releases. Depending on the version of Chef you are using within your organization may dictate which version of Ohai is being used.



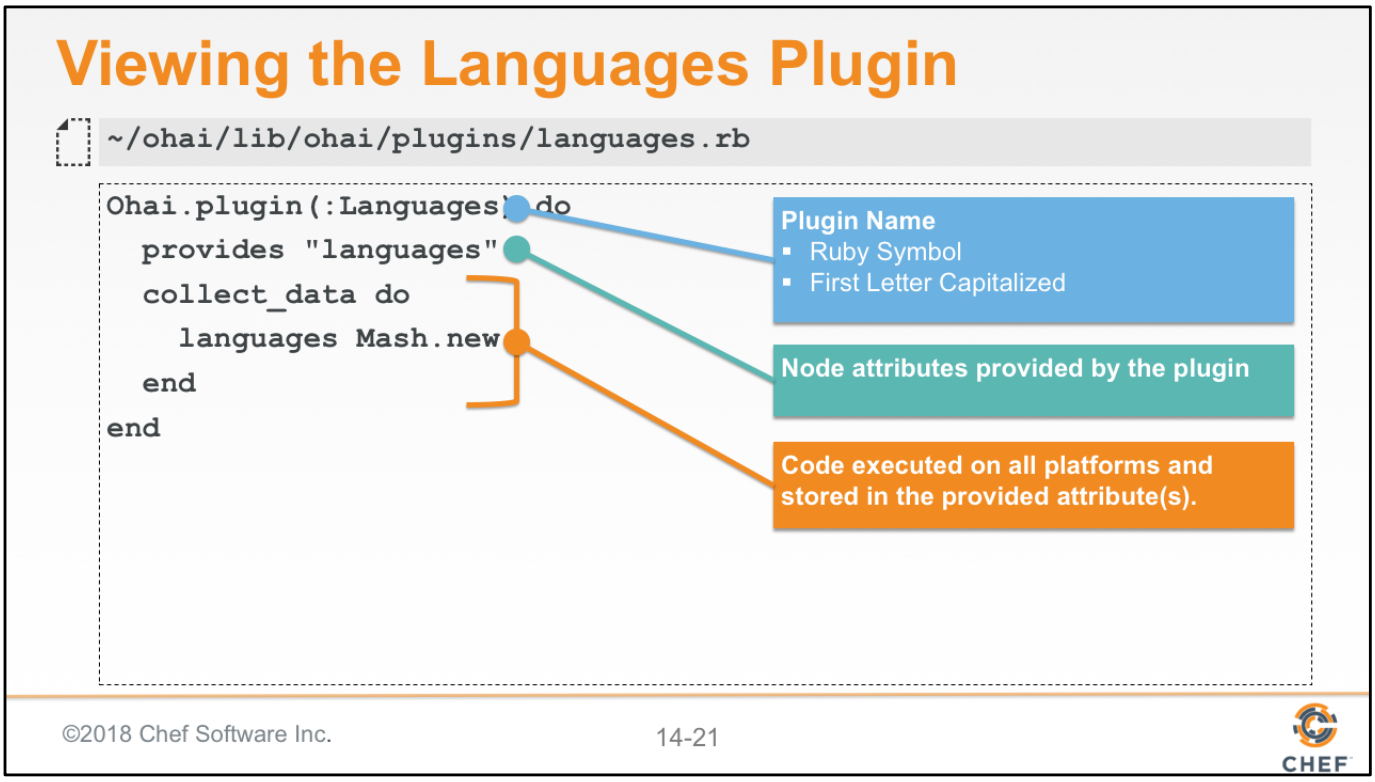
Ohai 6 introduced the ability to express plugins through a DSL. Ohai 7 refined that DSL. Ohai 8 continues to use that same language. The following slides and our exercise in the next module will focus on the DSL defined in Ohai 7.

Viewing the Languages Plugin

~/ohai/lib/ohai/plugins/languages.rb

```
Ohai.plugin(:Languages) do
  provides "languages"
  collect_data do
    languages Mash.new
  end
end
```


Let's load the languages plugin and review the basic structure of the plugin.



A plugin starts with invoking a method on the Ohai class with a single parameter. That parameter provided is the symbol name of the plugin. All Ohai plugins must have a symbol name with the first letter capitalized.

The remainder of the plugin is defined within the block of the 'plugin' method. The 'provides' method specifies what attribute or attributes the plugin will be added to the node object. The 'collect_data' method defines a block which contains the code that is executed on all platforms. This block of code will often times set the values of the attributes the plugin provides.

Viewing the Language Plugin

~/ohai/lib/ohai/plugins/languages.rb

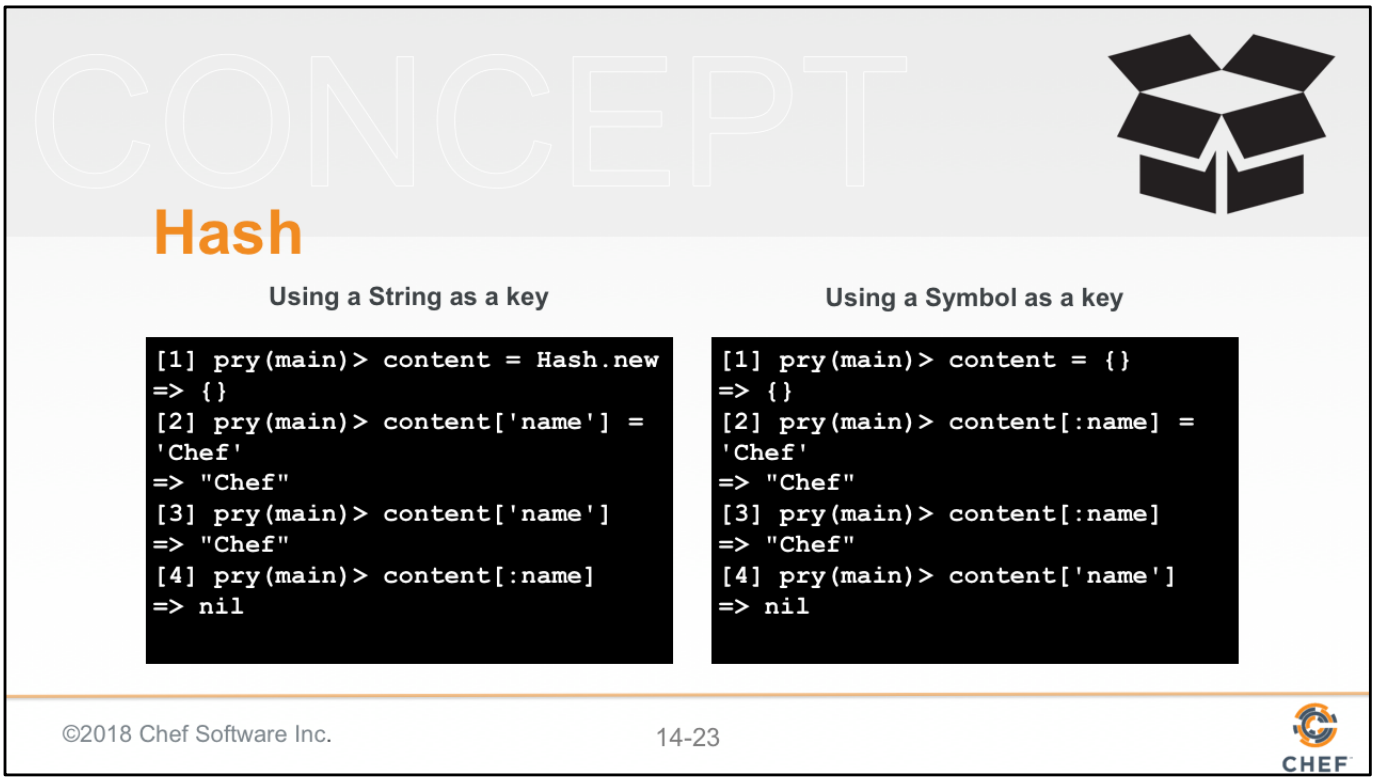
```
Ohai.plugin(:Languages) do
  provides "languages"
  collect_data do
    languages Mash.new
  end
end
```

The Languages plugin provides the node attribute 'languages' which is populated, on all platforms, with a Mash.

CHEF

This plugin is named Languages. It provides the languages attribute on the node. This languages attribute is populated with the contents of a new Mash.

But what is a Mash?



To understand what a Mash is first let's talk about Ruby's Hash. Hashes allow you to store values with a key; often times these keys are Ruby Strings or Ruby Symbols. When you want to retrieve that value you need to provide the same key. So if say you stored data with a Symbol key it is only retrievable with a Symbol key. The same could be said for using a String key.

CONCEPT

Mash

Using a String as a key

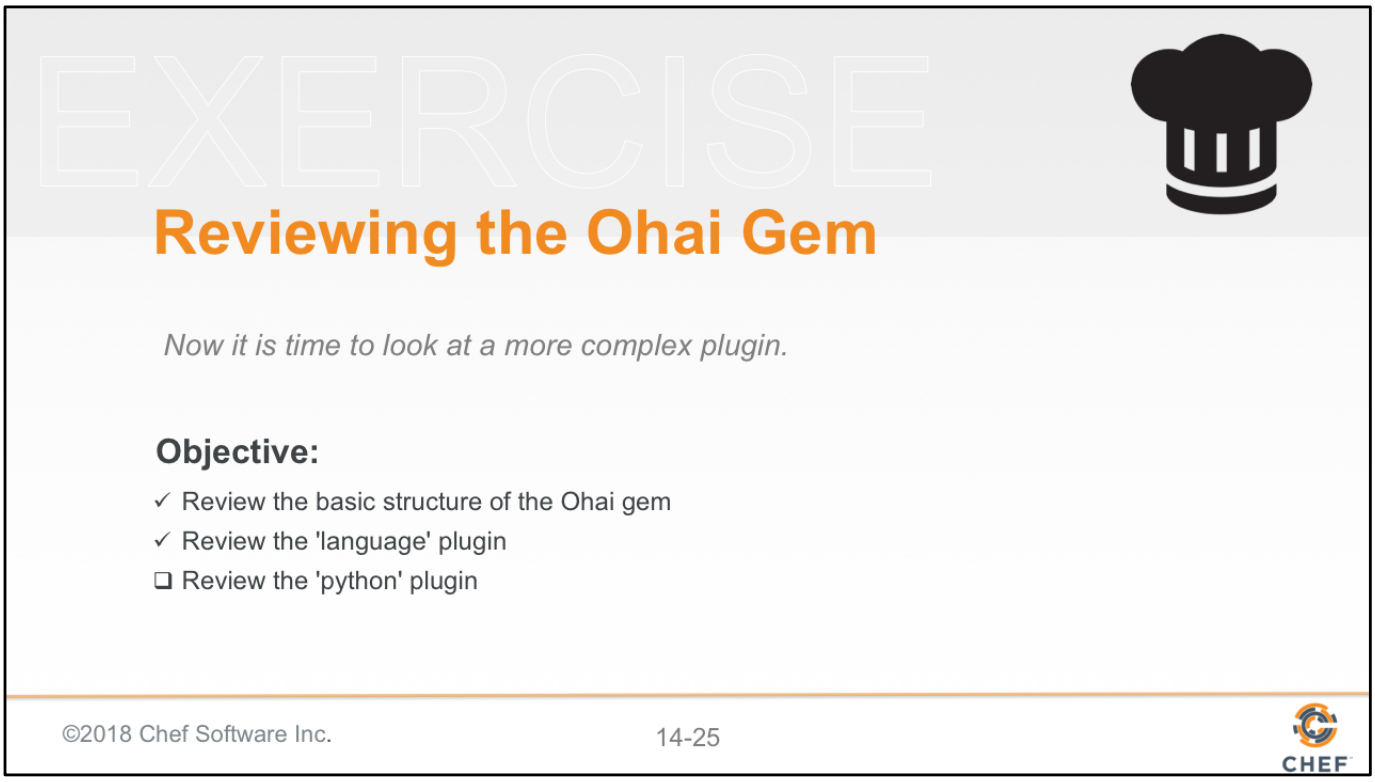
```
[1] pry(main)> require 'chef'
=> true
[2] pry(main)> content = Mash.new
=> {}
[3] pry(main)> content['name'] = 'Chef'
=> "Chef"
[4] pry(main)> content['name']
=> "Chef"
[5] pry(main)> content[:name]
=> "Chef"
```

Using a Symbol as a key

```
[1] pry(main)> require 'chef'
=> true
[2] pry(main)> content = Mash.new
=> {}
[3] pry(main)> content[:name] = 'Chef'
=> "Chef"
[4] pry(main)> content[:name]
=> "Chef"
[5] pry(main)> content['name']
=> "Chef"
```


CHEF

A Mash is similar to a Ruby Hash except that it is indifferent to whether you provide it a String key or Symbol key. Either of those types of keys will return value stored by the other. This more lenient data structure allows for these two keys to be used interchangeably. Allowing us to use whichever key style we prefer without being penalized if we were to guess the key style that differs from other plugins.



The language plugin is small plugin that setups up a data structure for other language plugins to add more information to it. Let's review a specific language plugin to see a more complex implementation.

Viewing the Python plugin

~/ohai/lib/ohai/plugins/python.rb

```
Ohai.plugin(:Python) do
  provides "languages/python"

  depends "languages"

  collect_data do
    begin
      so = shell_out("python -c \"import sys; print (sys.version)\"")
      # Sample output:
      # 2.7.11 (default, Dec 26 2015, 17:47:53)
      # [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
      if so.exitstatus == 0
        python = Mash.new
        output = so.stdout.split
        python[:version] = output[0]
        if output.length >= 6
```

Node attributes provided by the plugin

This plugin depends on the attributes in the Languages to be defined

CHEF

Here within the Python plugin we see the same structure with a dependency and a significant amount of work being done in the 'collect_data' method block. The attribute provided by this plugin can be found on the node object under the specified path. Remember this is the same path structure you use on the command-line when wanting to traverse the attributes provided.

The dependency described here states that this plugin requires that the node attribute value 'languages' must be defined first before this plugin will execute. Ohai will determine how to execute the plugins based on these dependencies.

Viewing the Python plugin

~/ohai/lib/ohai/plugins/python.rb

```
Ohai.plugin(:Python) do
  provides "languages/python"

  depends "languages"

  collect_data do
    begin
      so = shell_out("python -c \"import sys; print (sys.version)\"")
      # Sample output:
      # 2.7.11 (default, Dec 26 2015, 17:47:53)
      # [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
      if so.exitstatus == 0
        python = Mash.new
        output = so.stdout.split
        python[:version] = output[0]
        if output.length >= 6
```


Within the collect_data block we use a helper method named 'shell_out'. This 'shell_out' method accepts a single parameter which is the command to run. This 'shell_out' method will generate an object for which you can ask for the standard output, standard error, and the exit status.

This command is executed and if the status is successful (0 status code) then look at the standard output, split it into multiple lines, extract the version and possibly any build date information, and then store that information into the Mash that was created by the Languages plugin. If a failure occurs at any point catch that error and display a debug message.

You will find that most Ohai plugins will fit the following pattern. Perform a system related call to collect some data, use Ruby to process that data, and then store the data.

CONCEPT

collect_data for a specific Platform

Plugins can collect data in different ways across different platforms. When defining a collect_data block if you do not provide any arguments it is assumed the default and all platforms unless you define a collect_data block specific for a platform.

CHEF

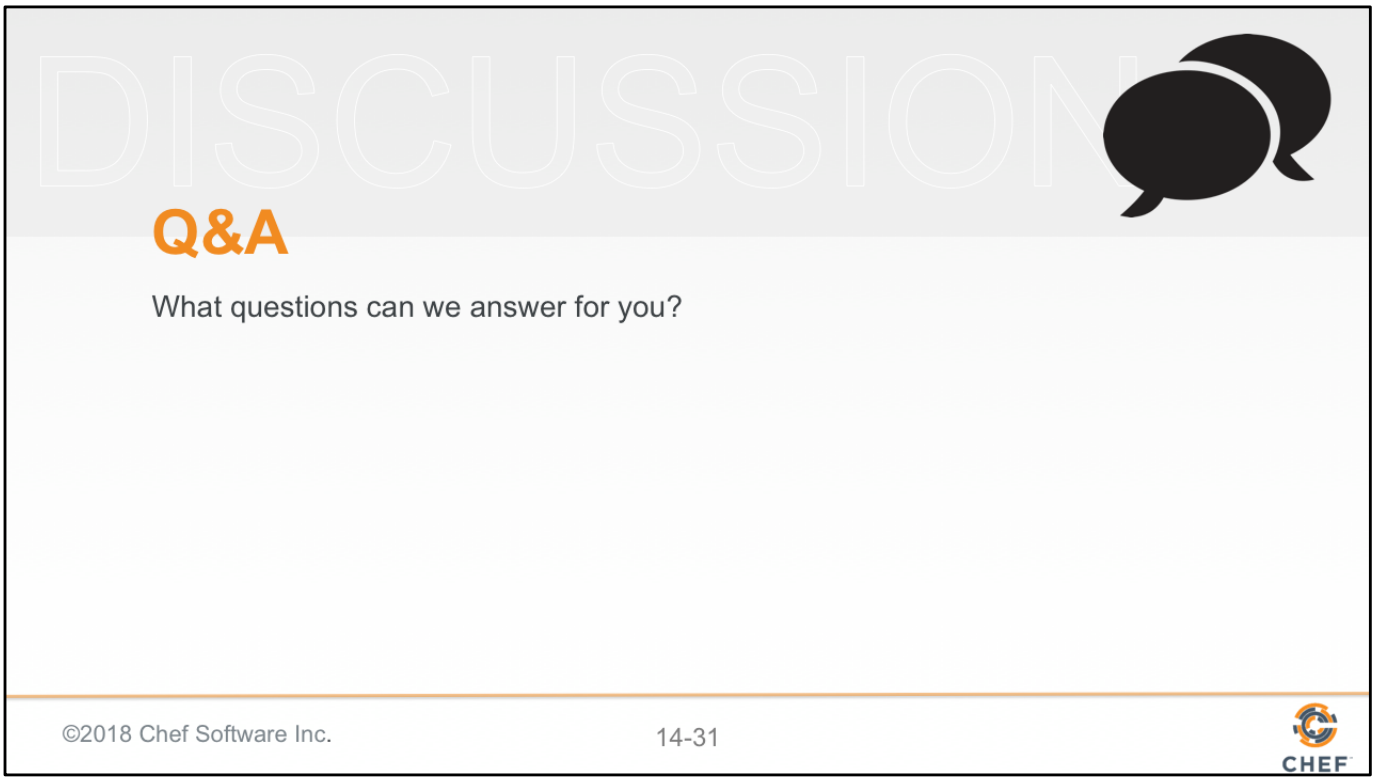
We were able to view the contents of the gem and examine the contents of a few plugins to give us an understanding of how plugins are structured. Now it is time for use to create our own.

Discussion

How are the structures of a Rubygem and a Cookbook similar to each other?

What are the requirements when specifying the name of a Ohai plugin?

What is the difference between a Ruby Hash and a Mash?



What questions can we answer for you?


CHEF™