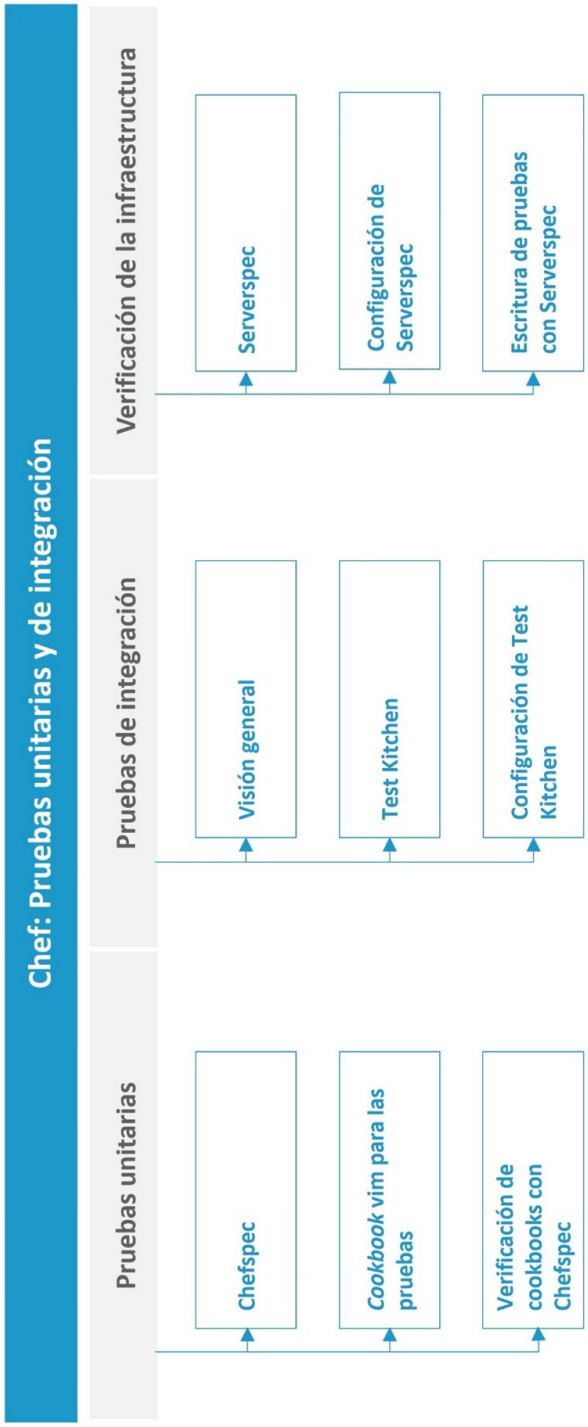
Herramientas de Automatización de Despliegues

Tema 4. Chef. Pruebas unitarias y de integración

Índice

Esquema



4.1. Introducción y objetivos

Por medio de la utilización de herramientas de gestión de la configuración tales como Chef para gestionar tu infraestructura puedes incorporar las mejores prácticas empleadas en el desarrollo de aplicaciones y aplicarlas así al desarrollo y la mejora de la infraestructura de servidores.

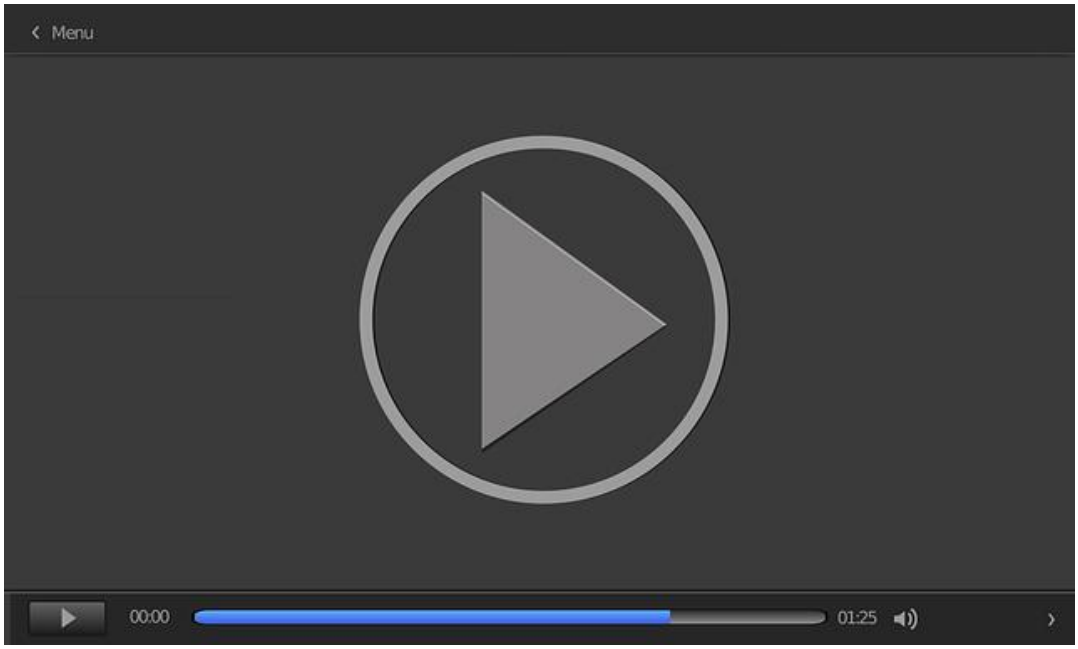
Estas mejores prácticas incluyen, aparte de utilizar un sistema de control de versiones para almacenar y gestionar tus ficheros de definición de la configuración que componen los *cookbooks,* definir una serie de pruebas para verificar que la configuración definida realmente hace lo que se espera, y que futuros cambios en la configuración no producen efectos no deseados en los recursos ya existentes (regresión). Estos conjuntos de pruebas se catalogan en distintos grupos, según su alcance, tales como pruebas unitarias, que se enfocan en verificar la funcionalidad de un elemento individualmente, o pruebas de integración, enfocadas en verificar cómo interactúa un conjunto de elementos relacionados entre sí.

Los objetivos de este tema son los siguientes:

- Realizar pruebas unitarias mediante ChefSpec.
- Realizar pruebas de integración con Test Kitchen.
- Verificar la ejecución con Serverspec.

Ideas clave

Aquí puedes ver el vídeo *Mejores prácticas con Chef*:



Accede al vídeo:https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=68f2c9b6-0907-4f89-a0ac-abb6011c3ea7

4.2. Pruebas unitarias de cookbooks con ChefSpec

El hecho de tratar la infraestructura como código nos abre la posibilidad de implementar pruebas unitarias automatizadas para verificar la funcionalidad de configuración requerida. En este apartado vamos a ver cómo se puede utilizar ChefSpec con Chef para este fin.

Las metodologías de desarrollo ágiles trajeron ideas innovadoras que mejoraron y enriquecieron el mundo del desarrollo de *software,* cambiando la manera de verlo y de hacer las cosas. Esta misma filosofía ágil también es aplicable a la forma en que vemos nuestra infraestructura.

Al gestionar la infraestructura de la misma manera que otras partes importantes del proyecto, nos surge la necesidad de que la configuración de dicha infraestructura sea sometida a pruebas con el fin de verificar y garantizar su fiabilidad. Gracias a herramientas como Chef y similares de gestión de la configuración, estas ideas se pueden implementar de una manera similar a como se implementan en la gestión del propio código fuente de las aplicaciones.

Una de las pautas fundamentales para la creación de software sostenible y de calidad es **poder hacer pruebas** de forma automatizada. En función de este *testing,* podemos realizar cambios y mejoras en el código en el futuro con la garantía de que, si algo de lo ya existente deja de funcionar, las pruebas fallarán y nos avisarán a este respecto.

En general, la manera fundamental y más sencilla para empezar con las pruebas es la **prueba o el test unitario.** Vamos a comenzar repasando los principios básicos de la prueba unitaria con Chef y para ello vamos a utilizar ChefSpec, que es un *framework* de pruebas para Chef construido sobre RSpec, herramienta de pruebas de Ruby.

Prerrequisitos

Dado que estamos trabajando con el software de gestión de infraestructura Chef, debemos contar con todas las herramientas del ecosistema Chef instaladas para pruebas, mediante la instalación del paquete Chef Workstation o el kit de desarrollo Chef. Esto nos proporciona, aparte de las herramientas esenciales para gestionar el código que ejecuta la infraestructura, las herramientas de pruebas como ChefSpec, que vamos a utilizar para las pruebas unitarias.

Si necesitas más detalles sobre la configuración del entorno de desarrollo Chef, consulta el tema «Chef: introducción e instalación».

Escritura de un *cookbook* para las pruebas

Vamos a comenzar por escribir algunas recetas básicas, para poder utilizarlas con ChefSpec y poder así ir viendo más profundamente lo que podemos hacer con esta herramienta. Para ello, vamos a escribir un *cookbook* que va a instalar el editor Vim en Linux, así como algunos paquetes adicionales. En primer lugar, vamos a necesitar generar un *cookbook* básico, ubicado en nuestro directorio de *cookbooks* mediante:

```
chef generate <em>cookbook</em> vim_pruebas_chef
```

Nuestro libro de cocina recién generado debería tener este aspecto:

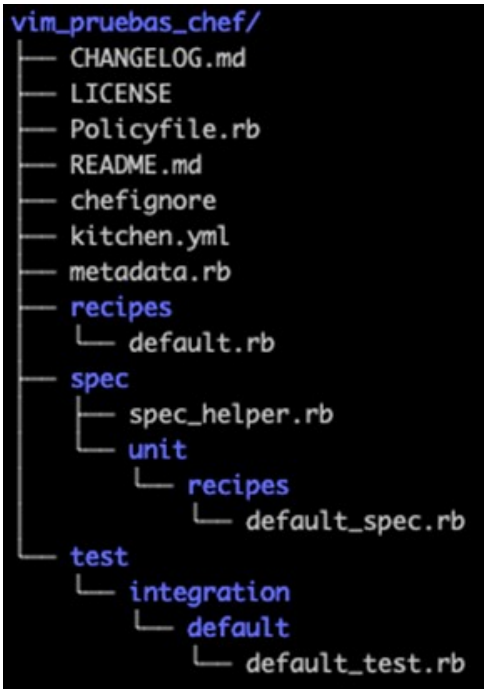


Figura 1. Estructura de directorios y ficheros de un *cookbook*. Fuente: elaboración propia.

A continuación, vamos a escribir algunas recetas que se aseguren de que Vim esté instalado, y usaremos las fuentes o los paquetes oficiales. Incluimos lo siguiente en el fichero `recipes/default.rb` dentro del *cookbook*:

```
begin
```

```
 include_recipe "vim::#{node['vim']['install_method']}" rescue
Chef::Exceptions::RecipeNotFound
```

```
 Chef::Log.warn"A #{node['vim']['install_method']} recipe does not
exist for the platform_family: #{node['platform_family']}"
```

```
end
```

En esta receta hemos incluido otra receta en función de un atributo que indica el método de instalación (`node['vim']['install_method']`), captura la excepción si

no existe una receta para el método de instalación especificado y muestra un mensaje al respecto.

Ahora, la manera más adecuada de soportar diversas plataformas con diferentes nombres de paquetes desde Chef es mediante la utilización de la función de ayuda `value_for_platform` que, a partir de una estructura de datos, nos devuelve un valor correspondiente a la plataforma sobre la que ejecuta la receta. Así, en función de este valor, podremos determinar los paquetes que deberemos instalar en cada plataforma. Incluimos lo siguiente en el fichero `recipes/package.rb`:

```
vim_base_pkgs = value_for_platform({

 ["ubuntu","debian","arch"] => {"default" => ["vim"]},

 ["redhat","centos","fedora","scientific"] => {"default"=> ["vim-
minimal","vim-enhanced"]},

 "default"=> ["vim"]})

vim_base_pkgs.each do |vim_base_pkg|

package vim_base_pkg

end

node['vim']['extra_packages'].each do |vimpkg|

package vimpkg

end
```

La instalación a partir del código fuente de vim también sería sencilla ya que, aparte de instalar las dependencias básicas necesarias, lo único que se debe hacer en la receta es descargar la versión deseada del código fuente, y ejecutar el comando

`make`. Creamos la receta `recipes/source.rb` que realizará las acciones indicadas con el siguiente contenido:

```
cache_path = Chef::Config['file_cache_path']

source_version = node['vim']['source']['version']

apt_update "update_apt_cache"

node['vim']['source']['dependencies'].each do |dependency|

 package dependency do

 action :install

 end

end

remote_file "#{cache_path}/vim-#{source_version}.tar.bz2" do

 source "http://ftp.vim.org/pub/vim/unix/vim-#
{source_version}.tar.bz2"

 checksum node['vim']['source']['checksum']

 notifies :run, "bash[install_vim]", :immediately

end

bash "install_vim" do

 cwd cache_path

 code <<-EOH
```

```
mkdir vim-#{source_version}
```

```
tar -jxf vim-#{source_version}.tar.bz2 -C vim-#{source_version} --
strip-components 1
```

```
(cd vim-#{source_version}/ && ./configure #{node['vim']['source']
['configuration']} && make && make install)
```

```
EOH
```

```
 action :nothing
```

```
end
```

Estas serían las recetas necesarias, a las que debemos complementar con un conjunto de atributos predeterminados para cada caso que nos proporcionen los datos requeridos para que el *cookbook* funcione. A continuación, se muestra el contenido de los ficheros de atributos para nuestro caso.

```
#./attributes/default.rb
```

```
default['vim']['extra_packages'] = []
```

```
default['vim']['install_method'] = 'package'
```

```
#./attributes/source.rb
```

```
default['vim']['source']['version'] = '8.2'
```

```
default['vim']['source']['checksum'] =
'f087f821831b4fece16a0461d574ccd55a8279f64d635510a1e10225966ced3b'
```

```
default['vim']['source']['prefix'] = '/usr/local'
```

```
default['vim']['source']['configuration'] = "--without-x --enable-
```

```
pythoninterp --enable-rubyinterp --enable-tclinterp --enable-
luainterp --enable-perlinterp --enable-cscope --with-features=huge
--prefix=#{default['vim']['source']['prefix']}"

default['vim']['source']['dependencies'] =

 if platform_family? 'rhel', 'fedora'

 %w( ctags

 gcc

 lua-devel

 make

 ncurses-devel

 perl-devel

 perl-ExtUtils-CBuilder

 perl-ExtUtils-Embed

 perl-ExtUtils-ParseXS

 python-devel

 ruby-devel

 tcl-devel

 bzip2

 )
```

```
elsif platform_family?('suse')

%w( ctags

gcc

lua-devel

make

ncurses-devel

perl

python-devel

ruby-devel

tcl-devel

tar

)

else

%w( exuberant-ctags

gcc

libncurses5-dev

libperl-dev

lua5.1
```

```
make

python-dev

ruby-dev

tcl-dev

bzip2

)

end
```

Pero ¿cómo asegurarnos de que el código de estas recetas junto con sus atributos realmente funciona o si es un código válido?

Por supuesto, podríamos ejecutar el *cookbook* en un servidor existente o en una máquina virtual e inspeccionar los resultados manualmente. Sin embargo, a medida que se empieza a trabajar con más de un par de *cookbooks*, se hace un trabajo pesado y laborioso, haciendo que el proceso de verificación sea lento y doloroso. Para afrontar esta circunstancia, sería más adecuado escribir una serie de pruebas unitarias que puedan verificar que los métodos apropiados de Chef se ejecutan, en lugar de tener que verificar los resultados del Chef *run* manualmente.

En cualquier caso, cabe señalar que las pruebas unitarias no sustituyen a las pruebas de integración, sino que las complementan y son mucho mejor que no tener nada, además de que son más sencillas de elaborar. Estas pruebas unitarias pueden detectar errores y *bugs* en una fase de desarrollo temprana y, por tanto, pueden ahorrarnos un valioso tiempo sin generar además ningún coste de aprovisionamiento.

Verificación de *cookbooks* con pruebas ChefSpec

La manera de comenzar a escribir pruebas unitarias para un *cookbook* es crear en la carpeta `spec` un archivo de especificaciones correspondiente a cada receta, por lo que vamos a empezar escribiendo las especificaciones para la receta predeterminada. Dado que esta receta solo consta de la inclusión de otra receta para cada método de instalación, empezar nuestras pruebas con ella es lo más conveniente.

Lo primero que debemos hacer es incluir la gema de ChefSpec y establecer un run en memoria, lo que ejecutará muy rápido al no estar aprovisionando realmente ningún equipo. Esto es fácil de configurar, incluyendo lo siguiente en el fichero `spec/unit/recipes/default_spec.rb` o, si ya existe el archivo (las últimas versiones de Chef incorporan ya alguna prueba unitaria de ejemplo), sustituiríamos el contenido del bloque describe por el siguiente:

```
require 'chefspec'

describe 'vim_pruebas_chef::default' do

 platform 'ubuntu'

end
```

El siguiente paso será definir algunos casos de prueba de la funcionalidad que se describe en el interior de la receta y establecer las expectativas de estos casos de prueba.

Lo más básico que podemos probar es que nuestra configuración de ejecución Chef establezca un atributo predeterminado para especificar el método de instalación. Para ello, hay que agregar un nuevo bloque `it` a la receta de pruebas, dentro del bloque describe:

```
require 'chefspec'

describe 'vim_pruebas_chef::default' do

 platform 'ubuntu'

 context 'with default attributes' do

 it "should have default install_method 'package'" do

 expect(chef_run.node['vim']['install_method']).to eq('package')

 end

 end

end
```

Vamos ahora a ejecutar los test mediante el siguiente comando Chef, que debería mostrar un resultado similar al que se muestra.

```
$ chef exec rspec

.

Finished in 3.28 seconds (files took 2.35 seconds to load)

1 example, 0 failures
```

Tal como podemos apreciar, nuestro caso ha pasado la prueba sin errores, verificando así que el valor por defecto del atributo está configurado. A continuación, vamos a comprobar si la receta que se incluye en la ejecución es la adecuada. Añadimos el siguiente caso de prueba dentro del bloque `context` del fragmento anterior:

```
 it "should include the vim_pruebas_chef::package recipe when
install_method='package'" do

 expect(chef_run).to include_recipe('vim_pruebas_chef::package')

 end
```

Y creamos un nuevo bloque `context` dentro del bloque padre describe para representar el caso de un valor diferente para el atributo `install_method`, cuyo valor sobrescribimos al inicio del bloque:

```
context "with 'source' as install_method" do

 override_attributes['vim']['install_method'] = 'source'

 it "should include the vim_pruebas_chef::source recipe when
install_method='source'" do

 expect(chef_run).to include_recipe('vim_pruebas_chef::source')

 end

 end
```

Además de las comprobaciones *(matchers)* que ya incluye RSpec por defecto, ChefSpec define *matchers* adicionales para todos los recursos básicos de Chef.

Para consultar los RSpec matchers estándar:

https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

Para consultar los matchers propios de Chef:

https://www.rubydoc.info/github/sethvargo/chefspec

A continuación, se muestra el fichero `spec/unit/recipes/default_spec.rb` de pruebas unitarias completo que hemos definido en los pasos anteriores con los tres casos de prueba definidos, que deberían pasar con éxito:

```
require 'spec_helper'

describe 'vim_pruebas_chef::default' do

 platform 'ubuntu'

 context 'with default attributes' do

 it "should have default install_method 'package'" do

 expect(chef_run.node['vim']['install_method']).to eq('package')

 end

 it "should include the vim_pruebas_chef::package recipe when
install_method='package'" do

 expect(chef_run).to include_recipe('vim_pruebas_chef::package')

 end

 end

 context "with 'source' as install_method" do

 override_attributes['vim']['install_method'] = 'source'

 it "should include the vim_pruebas_chef::source recipe when
install_method='source'" do

 expect(chef_run).to include_recipe('vim_pruebas_chef::source')
```

```
 end

 end

end
```

Dado que nuestras pruebas están escritas en lenguaje Ruby al igual que los propios *cookbooks*, podemos también generar dinámicamente los casos de prueba para múltiples plataformas utilizando código Ruby. Vamos entonces ahora a probar nuestra receta `package.rb` contra diferentes versiones de Ubuntu, Debian y RedHat Linux. Creamos el fichero de pruebas `spec/unit/recipes/package_spec.rb` con:

```
require 'spec_helper'

describe 'vim_pruebas_chef::package' do

 package_checks = {

 'ubuntu' => {

 '16.04' => ['vim'],

 '18.04' => ['vim'],

 '20.04' => ['vim'],

 },

 'debian' => {

 '8.11' => ['vim'],

 '9.11' => ['vim'],
```

```
'10' => ['vim']

},

'redhat'=> {

'7.7'=> ['vim-minimal','vim-enhanced'] ,

'8'=> ['vim-minimal','vim-enhanced']

}

}

package_checks.each do |platform, versions|

versions.each do |version, packages|

packages.each do |package_name|

it "should install #{package_name} on #{platform} #{version}" do

chef_run = ChefSpec::SoloRunner.new(platform: platform, version:
version, file_cache_path: '/var/chef/cache') do |node|

node.normal['vim']['install_method'] = 'package'

end.converge(described_recipe)

expect(chef_run).to install_package(package_name)

end

end
```

```
end
```

```
end
```

```
end
```

En este ejemplo se itera por la lista de plataformas y versiones, y se generan los casos de prueba dinámicamente en función de ellos, cada uno define las expectativas de los paquetes necesarios que se instalarán.

A continuación, se enumeran las herramientas que hemos utilizado hasta aquí. En el siguiente apartado vamos a avanzar en nuestro plan de pruebas utilizando herramientas adicionales para definir otro tipo de pruebas:

- Chef Workstation / ChefDK
- RSpec
- ChefSpec

4.3. Pruebas de integración de infraestructura con Test Kitchen

Para las pruebas de integración de infraestructura, vamos a ver cómo probar automáticamente los *cookbooks* de Chef en una máquina virtual que es una copia del servidor de producción.

Uno de los mayores impedimentos a la hora de gestionar la infraestructura como código es la forma en que la probamos, debido a que la práctica de escribir pruebas automatizadas para la infraestructura es algo que en el pasado no existía y, por consiguiente, el proceso se realizaba de manera manual. Este proceso resultaba ser repetitivo y propenso a errores, al tener que iniciar sesión en una máquina virtual y ejecutar varios comandos manualmente para verificar el sistema.

El principal inconveniente de este proceso manual es que se requiere contar con una lista concreta de todo lo que debe ser probado y, si a esto añadimos un equipo de personas que está constantemente modificando y evolucionando el sistema, esto se vuelve muy complicado. Por otra parte, aunque se cuente con una batería de pruebas completa y bien definida, si estas son difíciles de ejecutar y no están automatizadas, el equipo terminará habitualmente haciendo caso omiso de ellas y saltándose su ejecución.

Gracias al desarrollo de las herramientas de gestión de la configuración, tales como Ansible, Chef o Puppet, el proceso de las pruebas ha evolucionado considerablemente. El hecho de tener herramientas que facilitan la automatización de las pruebas hace que estas se vuelvan más fáciles de ejecutar, más fiables y, lo que es más importante, mucho más rápidas y sin apenas requerir intervención manual.

En este apartado vamos a cubrir el desarrollo de pruebas de integración para los

servidores administrados por Chef, para lo que utilizaremos la herramienta Test Kitchen como ejecutor de máquinas virtuales, respaldado por Vagrant y VirtualBox. La prueba propiamente dicha la escribiremos en Ruby utilizando el *framework* de pruebas Serverspec.

Requisitos previos

Todas las herramientas que vamos a manejar están incluidas en la instalación de Chef WorkStation o ChefDK (kit de desarrollo Chef), por lo que ya deberías tenerlas instaladas. Si no es así, consulta el tema «Chef: introducción e instalación» para revisar el apartado de instalación.

Las demás herramientas de ayuda que vamos a utilizar, tales como Vagrant y VirtualBox, también deberían estar instaladas si has seguido los ejemplos de los temas anteriores, por lo que ya deberías estar en disposición de poder desarrollar las pruebas de integración mediante Test Kitchen.

Visión general de las pruebas de integración

Para el propósito de este apartado vamos a utilizar el *cookbook* básico de instalación de Vim desde el código fuente o desde los paquetes oficiales de distribución, que ya hemos utilizado en el apartado anterior. De cara a las pruebas de integración, lo que más nos interesa son los resultados de la propia ejecución, mientras que la implementación concreta es menos relevante.

Vamos a probar los dos escenarios diferentes posibles, el que incluye la receta para la instalación de Vim desde paquetes y el que lo instalará desde las fuentes.

Test Kitchen

Test Kitchen es una herramienta de automatización de pruebas distribuida con Chef WorkStation y ChefDK que es capaz de gestionar máquinas virtuales internamente llamadas nodos, aplicar la configuración Chef y realizar test. Se puede utilizar de una manera totalmente automatizada, donde todos los pasos se ejecutan

secuencialmente, incluyendo la destrucción del nodo al final de la ejecución, aunque también se puede optar por la ejecución manual de tareas.

De cara a ejecutar las pruebas de integración, es necesario utilizar la misma configuración Chef que se va a utilizar en un entorno real, con la misma lista de ejecución, recetas, funciones y atributos. Opcionalmente, puede ser requerido el proporcionar una serie de atributos adicionales personalizados, que se utilizarán solo en el entorno de prueba, como pueden ser datos de prueba, por ejemplo.

Un nodo se puede representar en Test Kitchen con cualquier tipo de virtualización, a través de *plugins* denominados ***drivers.*** En la mayoría de los casos se suele utilizar Vagrant, pero hay diferentes alternativas tales como Docker o cualquiera de los muchos proveedores de nube soportados, como Amazon Web Services o DigitalOcean.

La lista completa de los drivers soportados por Test Kitchen se puede encontrar en: https://kitchen.ci/docs/drivers/

Todos los ajustes de configuración de Test Kitchen se definen dentro del fichero `kitchen.yml` en el directorio raíz del *cookbook*.

A pesar de que en nuestro caso vamos a utilizar Test Kitchen con un solo *cookbook*, se puede utilizar para probar el funcionamiento de la lista completa que esté asociada con el tipo de máquina que estés utilizando.

Configuración de Test Kitchen

Vamos a probar nuestro *cookbook* `vim_pruebas_chef` y sus recetas mediante `chef_solo` en las plataformas Ubuntu 18.04 y CentOS 7.8 con el *driver* de Vagrant. El fichero de configuración `kitchen.yml` debe contener lo siguiente:

```
---
```

```
driver:

 name: vagrant

provisioner:

 name: chef_zero

platforms:

 - name: ubuntu-18.04

 - name: centos-7.8

suites:

- name: source

 run_list:

 - recipe[vim_pruebas_chef::default]

 attributes:

 vim:

 install_method: "source"

- name: package

 run_list:

 - recipe[vim_pruebas_chef::default]

 attributes:
```

```
vim:

install_method: "package"
```

Tal como podemos ver, en el fichero YAML estamos especificando el *driver*, el aprovisionador, las plataformas y las suites de pruebas, que en nuestro caso se corresponde a dos elementos para probar nuestro *cookbook* con los dos posibles valores de `install_method`. Para verificar la configuración y ver un resumen de todas las instancias disponibles proporcionadas por Test Kitchen, basta con ejecutar:

```
$ kitchen list

Instance Driver Provisioner Verifier Transport Last Action Last
Error

source-ubuntu-1804 Vagrant ChefZero Busser Ssh <Not Created>
<None>

source-centos-78 Vagrant ChefZero Busser Ssh <Not Created> <None>

package-ubuntu-1804 Vagrant ChefZero Busser Ssh <Not Created>
<None>

package-centos-78 Vagrant ChefZero Busser Ssh <Not Created> <None>
```

Como se puede ver, Test Kitchen nos está proporcionando cuatro casos: cada suite contra cada plataforma definida en el archivo de configuración. Estas instancias no se han creado aún, así que vamos a escribir algunos casos de prueba para poder probarlos contra estas instancias en el siguiente apartado.

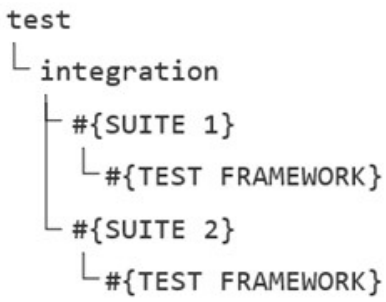
4.4. Verificación de la ejecución de Chef con Serverspec

Ahora que tenemos preparada la infraestructura completa para realizar nuestras pruebas de integración, nos falta establecer nuestras expectativas mediante los casos de prueba y encontrar una manera de verificarlos, para lo cual necesitaremos otra herramienta, dado que **Test Kitchen** es únicamente una herramienta de automatización, y no nos proporciona una manera de escribir los casos de prueba.

Serverspec es un *framework* de pruebas basado en RSpec, que es a su vez un *framework* de pruebas muy extendido entre la comunidad Ruby. Permite escribir pruebas de estilo RSpec para comprobar la configuración de la infraestructura, sin importar cómo ha sido aprovisionada dicha infraestructura. Las pruebas se definen de una manera descriptiva, así que con herramientas como esta se reduce al mínimo la posibilidad del error humano.

Configuración de Serverspec

Para poder comenzar a escribir pruebas con Serverspec, es necesario contar con un subdirectorio llamado `integration` dentro del directorio de pruebas (test) de nuestro *cookbook*, donde se van a alojar todas las pruebas de integración. Bajo `integration` debería crearse un subdirectorio por cada suite, y dentro, a su vez, uno por cada *framework* de pruebas que vamos a utilizar, lo que significa que puedes utilizar tantos *frameworks* de pruebas como desees en la misma suite simultáneamente.



Dado que en nuestro caso vamos a utilizar Serverspec como el *framework* para las pruebas, a continuación se muestran los comandos necesarios para crear la estructura de directorios correspondiente:

```
mkdir -p test/integration/source/serverspec
```

```
mkdir -p test/integration/package/serverspec
```

Lo primero que vamos a necesitar es un *script* de apoyo hecho en Ruby que se encargue de cargar Serverspec y de establecer las opciones de configuración generales que vamos a necesitar, tales como la ruta utilizada para buscar los binarios durante la ejecución de las pruebas. Creamos el fichero `spec_helper.rb` dentro de la ruta `test/integration/package/serverspec/` con el siguiente contenido:

```
require 'serverspec'
```

```
require 'pathname'
```

```
set :backend, :exec
```

```
set :path, '/bin:/usr/local/bin:$PATH'
```

Una vez que contamos con este fichero de configuración, podemos incluirlo a través de la instrucción `require` al principio de cada fichero de pruebas que escribamos, con el objetivo de no tener que especificar esta configuración en cada nuevo fichero

de pruebas. Las pruebas se escriben en Ruby estándar, y por esto utilizamos la instrucción `require` que proporciona este lenguaje:

```
require 'spec_helper'
```

Escritura de pruebas con Serverspec

La prueba más básica que podemos realizar sobre nuestra receta es si nuestro aprovisionador ha instalado el paquete correspondiente que estamos esperando. Si la comprobación la hiciéramos de forma manual, lo más seguro es que solo ejecutásemos el comando `vim` para verificar si el programa existe y se ejecuta con éxito. Con Serverspec, tenemos una manera mucho más adecuada de comprobar el correcto funcionamiento de cualquier recurso específico, como puede ser un comando.

Además del uso de los comandos, con Serverspec también se puede verificar el estado propiamente dicho de los paquetes, lo cual es adecuado utilizar en nuestro caso concreto, al haber instalado en nuestro ejemplo los paquetes proporcionados por la distribución correspondiente. Las ventajas de utilizar Serverspec van mucho más allá, ya que el propio *framework* nos ofrece una gran cantidad de recursos y validaciones adicionales que permiten verificar una gran cantidad de elementos, tales como servicios o puertos.

Puedes encontrar la lista completa de los tipos de recursos que proporciona Serverspec con ejemplos en el sitio de la documentación Serverspec:
https://serverspec.org/resource_types.html

Por tanto, vamos a escribir nuestra primera prueba describiendo el estado del paquete `vim` e indicando que esperamos que se instale, añadiendo lo siguiente al fichero `test/integration/package/serverspec/vim_spec.rb`:

```
require 'spec_helper'
```

```
describe package('vim') do

 it {should be_installed}

end
```

De la misma manera podríamos también verificar cualquier otro recurso, como el *output* de un comando, o simplemente su estado de salida. En nuestro ejemplo, una comprobación que puede resultar útil es la versión que se ha instalado de la aplicación Vim. Del mismo modo que haríamos de forma manual, se puede comprobar la salida del comando `vim --version` y verificar si devuelve la versión deseada o incluye una cadena o expresión regular concreta, para lo cual añadimos lo siguiente a nuestro fichero `vim_spec.rb`:

```
describe command('vim --version') do

 its (:stdout) {should match /VIM - Vi IMproved/}

end
```

Dado que ya tenemos un par de casos de prueba definidos, vamos a ejecutarlos para comprobar su correcto funcionamiento, lo cual haremos mediante la ejecución del comando `kitchen test`, indicando como argumento el nombre de la instancia que se quiere probar. Ya habíamos visto cómo mostrar una lista de todas las instancias disponibles, mediante el comando `kitchen list`, por lo que vamos ahora a comprobar la instalación del paquete contra Ubuntu 18.04:

```
$ kitchen test package-ubuntu-1804

-----> Starting Test Kitchen (v2.5.1)

-----> Cleaning up any prior instances of <package-ubuntu-1804>
```

```
-----> Destroying <package-ubuntu-1804>...

 Finished destroying <package-ubuntu-1804> (0m0.00s).

-----> Testing <package-ubuntu-1804>

-----> Creating <package-ubuntu-1804>...

 Bringing machine 'default' up with 'virtualbox' provider...

 ==> default: Importing base box 'bento/ubuntu-18.04'...

[...]

-----> Running serverspec test suite

-----> Installing Serverspec..

[...]

 Package "vim"

 is expected to be installed

 Command "vim --version"

 stdout

 is expected to match /VIM - Vi IMproved/

 Finished in 0.15106 seconds (files took 0.27762 seconds to load)

 2 examples, 0 failures

 Downloading files from <package-ubuntu-1804>
```

```
Finished verifying <package-ubuntu-1804> (0m7.75s).

-----> Destroying <package-ubuntu-1804>...

==> default: Forcing shutdown of VM...

==> default: Destroying VM and associated drives...

Vagrant instance <package-ubuntu-1804> destroyed.

Finished destroying <package-ubuntu-1804> (0m5.76s).

Finished testing <package-ubuntu-1804> (1m7.30s).

-----> Test Kitchen is finished. (1m7.92s)
```

Como podemos ver en la salida de la ejecución, Test Kitchen ha creado una nueva máquina virtual con la versión y sistema operativo especificados, la ha aprovisionado con una lista específica de ejecución, ha ejecutado las pruebas contra ella y finalmente la ha destruido después de la ejecución, instalando en cada paso las herramientas requeridas, si fuera necesario. En nuestro caso, ha ejecutado los dos ejemplos que teníamos definidos sin errores.

Como ya hemos mencionado anteriormente, este es un proceso totalmente automatizado, pero para la fase de desarrollo sería demasiado lento tener que ejecutar este proceso con cada cambio de código, por lo que podemos crear (o utilizar el término *converge* en la terminología Test Kitchen) una instancia de forma manual y solo aplicar pruebas después de que se introduzca un cambio importante, por lo que no hay que esperar para hacer *converge* continuamente cada vez. Cuando las pruebas se pasan con éxito, vamos a destruir las instancias que no necesitamos.

Ahora vamos a probar a ejecutar manualmente el proceso de *converge* que se encarga de crear y aprovisionar la nueva instancia, y añadimos a continuación

algunos casos de prueba adicionales.

```
$ kitchen converge package-ubuntu-1804

-----> Starting Test Kitchen (v2.5.1)

-----> Creating <package-ubuntu-1804>...

 Bringing machine 'default' up with 'virtualbox' provider...

 ==> default: Importing base box 'bento/ubuntu-18.04'...

[...]

 Converging 1 resources

 Recipe: vim_pruebas_chef::package

 * apt_package[vim] action install (up to date)

 Running handlers:

 Running handlers complete

 Chef Infra Client finished, 0/1 resources updated in 02 seconds

 Downloading files from <package-ubuntu-1804>

 Finished converging <package-ubuntu-1804> (0m15.65s).

-----> Test Kitchen is finished. (1m3.13s)
```

Ya tenemos nuestra instancia aprovisionada, por lo que vamos ahora a incluir más casos de prueba para poder así incluir el soporte para múltiples plataformas que ya hemos definido en la fase de instalación. Tal como están definidas, las pruebas que

tenemos hasta ahora no son aplicables a otras plataformas donde el nombre del paquete no coincide con el nombre que tiene en Ubuntu, como ocurre por ejemplo en CentOS, que el paquete se llama `vim-minimal`. Dado que las pruebas se escriben en Ruby, se pueden utilizar las construcciones gramaticales de este lenguaje para establecer las condiciones en función de la familia de sistema operativo como, por ejemplo, para escribir casos de prueba separados para Ubuntu y CentOS. Modificamos el fichero `vim_spec.rb` con lo siguiente:

```
require'spec_helper'

if os[:family] == 'ubuntu'

describe package('vim') do

it {should be_installed}

end

end

if os[:family] =='redhat'

describe package('vim-minimal') do

it {should be_installed}

end

describe package('vim-enhanced') do

it {should be_installed}

end
```

```
end
```

```
describe command('vim --version') do
```

```
its (:stdout) {should match /VIM - Vi IMproved/}
```

```
end
```

Debido a que ya tenemos la instancia aprovisionada desde el paso anterior, se pueden aplicar nuevos cambios fácilmente para probarlos y comprobar el resultado de las pruebas.

```
kitchen verify package-ubuntu-1804
```

Las pruebas deberían pasar con éxito y, dado que ya no vamos a incluir ninguna otra mejora, vamos a proceder ahora a destruir el nodo:

```
kitchen destroy package-ubuntu-1804
```

Ya están hechas las pruebas para la instalación del editor desde paquetes, y se puede utilizar una técnica semejante para definir las pruebas de las recetas que compilan la aplicación desde el código fuente. Las pruebas que debemos definir tienen que comprobar si se ha instalado la versión adecuada y cuál es el usuario que la ha compilado.

Con vim esto se puede obtener con la salida del comando `vim --version`, que proporciona ambos datos, por lo que creamos el fichero `vim_spec.rb` en el directorio de la suite de las fuentes `test/integration/source/serverspec/` con lo siguiente:

```
require'spec_helper'
```

```
describe command('vim --version') do
```

```
 its(:stdout) {should match /VIM - Vi IMproved/}
```

```
 its(:stdout) {should match /Compiled by vagrant@source-/}

end
```

Para comprobar el resultado de las pruebas, las ejecutamos nuevamente de la forma estándar, indicando el tipo de nodo a crear:

```
kitchen test package-ubuntu-1804
```

Llegados a este punto, deberíamos poder ejecutar todas las pruebas contra todas las plataformas definidas inicialmente, mediante el comando `kitchen test`, sin proporcionar ningún parámetro, y Test Kitchen se encargará de generar, probar y destruir todos los nodos definidos en el fichero de configuración `kitchen.yml`.

Si en algún momento necesitas acceder a una instancia aprovisionada por Test Kitchen para, por ejemplo, probar un único comando, o explorar cómo verificar algo manualmente, puedes iniciar sesión en la instancia de convergencia, una vez que esté creada, y ejecutar comandos directamente. Para ello, puedes utilizar los siguientes comandos:

```
kitchen converge package-ubuntu-1804 # Crea la instancia de
convergencia
```

```
kitchen login package-ubuntu-1804
```

A modo de conclusión de todo lo que hemos visto hasta aquí, se puede decir que Test Kitchen nos posibilita de una manera sencilla la automatización del proceso de definir y probar la infraestructura de servidores heterogéneos. Con esta herramienta, se pueden ejecutar pruebas automatizadas contra la infraestructura de integración cuando ocurra algún cambio en la plataforma de infraestructura, modificación de código realizada por el equipo de desarrollo o incluso en el proceso de integración continua (CI), y se aplicarán así al código de la infraestructura todos los beneficios de

la automatización en la integración continua.

4.5. Referencias bibliográficas

Chef Software. (2020). *Chef Documentation*. https://docs.chef.io/

Marschall, M. (2015). *Chef Infrastructure Automation Cookbook*, Second Edition. Packt Publishing.

Waud, E. (2016). *Mastering Chef Provisioning*. Packt Publishing.

Sitio web de Serverspec

Miyashita, G. (2019). *Serverspec*. https://serverspec.org/

Este sitio web contiene todo lo relacionado con Serverspec, desde la documentación básica, tutorial, ejemplos, tipos de recursos, así como enlaces al repositorio de código fuente en GitHub.

Documentación de referencia ChefSpec

Chef Software. (2020). *ChefSpec*. https://docs.chef.io/workstation/chefspec/

Sitio de la documentación oficial de ChefSpec, el framework de pruebas de Chef basado en RSpec, que es a su vez un framework de pruebas BDD (Behavior-driven development, desarrollo basado en el comportamiento) para Ruby.

Repositorio GitHub de ChefSpec

GitHub. (2020). *ChefSpec*. https://github.com/chefspec/chefspec

Repositorio de código fuente de ChefSpec, donde puedes encontrar la última versión, revisar su documentación técnica, descargarte cualquier versión liberada o incluso hacer contribuciones de código.

Sitio oficial de Test Kitchen

KitchenCI. (2020). *Test Kitchen*. https://kitchen.ci/

Desde este sitio web puedes acceder a la documentación de referencia de Test Kitchen, con numerosos ejemplos y explicaciones sobre su funcionamiento, así como al repositorio de código fuente.

Test

1. ¿Cuál es la herramienta más adecuada en Chef para definir pruebas unitarias?

A. RSpec.

B. ChefSpec.

C. Test Kitchen.

D. Serverspec.

2. ¿Cuál es la herramienta más adecuada en Chef para automatizar las pruebas de integración?

A. RSpec.

B. ChefSpec.

C. Test Kitchen.

D. Serverspec.

3. ¿Cuál de las siguientes herramientas podemos utilizar en Chef para definir pruebas de integración?

A. RSpec.

B. Chef WorkStation.

C. Test Kitchen.

D. Serverspec.

4. ¿Cómo definimos una prueba unitaria en ChefSpec para una receta?

A. Creando en el mismo directorio que la receta un fichero `<receta>spec.rb`

B. Creando bajo el directorio test un fichero `<receta>spec.rb`

C. Creando en el mismo directorio que la receta un fichero con extension `.spec`

D. Creando bajo el directorio spec un fichero `<receta>spec.rb`

5. ¿Cómo se inicializa ChefSpec en un fichero de especificaciones de pruebas unitarias?

A. Incluyendo la gema de ChefSpec mediante `require 'chefspec'`

B. Incluyendo la gema de pruebas mediante `require 'test'`

C. Incluyendo la gema de pruebas unitarias mediante `require 'unittest'`

D. Son necesarias A y C.

6. ¿Cómo se comprueba en ChefSpec el resultado obtenido con el resultado esperado?

A. Mediante la instrucción `describe`

B. Mediante la instrucción `context`

C. Mediante la instrucción `expect`

D. Mediante la instrucción `verify`

7. ¿Cómo especifico en Test Kitchen los distintos sistemas operativos y versiones que deseo probar?

A. Mediante el bloque `systems` en `kitchen.yml`

B. Mediante el bloque `platforms` en `kitchen.yml`

C. Mediante el bloque `systems` en `test.yml`

D. Mediante el bloque `platforms` en `test.yml`

8. ¿Cómo defino en Test Kitchen las recetas que quiero probar?

A. Mediante el bloque `suites` en `kitchen.yml`

B. Mediante el bloque `recipes` en `kitchen.yml`

C. Mediante el bloque `suites` en `test.yml`

D. Mediante el bloque `recipes` en `test.yml`

9. ¿Cómo especifico mediante Serverspec una prueba sobre un recurso?

A. Mediante un bloque `resource`

B. Mediante un bloque `describe`

C. Mediante un bloque `suite`

D. Mediante un bloque `test`

10. ¿Cómo compruebo mediante Serverspec que la salida de un comando contiene una determinada cadena?

A. `describe (:stdout) {includes /VIM - Vi IMproved/}`

B. `its (:stdout) { includes /VIM - Vi IMproved/}`

C. `describe (:stdout) {should match /VIM - Vi IMproved/}`

D. `its (:stdout) {should match /VIM - Vi IMproved/}`