Configuration Management 101: Writing Puppet Manifests

digitalocean.com/community/tutorials/configuration-management-101-writing-puppet-manifests

Introduction

In a nutshell, server configuration management (also popularly referred to as IT Automation) is a solution for turning your infrastructure administration into a codebase, describing all processes necessary for deploying a server in a set of provisioning scripts that can be versioned and easily reused. It can greatly improve the integrity of any server infrastructure over time.

In a previous guide, we talked about the main benefits of implementing a configuration management strategy for your server infrastructure, how configuration management tools work, and what these tools typically have in common.

This part of the series will walk you through the process of automating server provisioning using Puppet, a popular configuration management tool capable of managing complex infrastructure in a transparent way, using a master server to orchestrate the configuration of the nodes. We will focus on the language terminology, syntax and features necessary for creating a simplified example to fully automate the deployment of an Ubuntu 14.04 web server using Apache.

This is the list of steps we need to automate in order to reach our goal:

1. Update the `apt` cache
2. Install Apache
3. Create a custom document root directory
4. Place an `index.html` file in the custom document root
5. Apply a template to set up our custom virtual host
6. Restart Apache

We will start by having a look at the terminology used by Puppet, followed by an overview of the main language features that can be used to write manifests. At the end of this guide, we will share the complete example so you can try it by yourself.

Note: this guide is intended to introduce you to the Puppet language and how to write manifests to automate your server provisioning. For a more introductory view of Puppet, including the steps necessary for installing and getting started with this tool, check our How To Install Puppet 4 in a Master-Agent Setup on Ubuntu 14.04 guide.

Getting Started

Before we can move to a more hands-on view of Puppet, it is important that we get acquainted with important terminology and concepts introduced by this tool.

Puppet Terms

- **Puppet Master**: the master server that controls configuration on the nodes
- **Puppet Agent Node**: a node controlled by a Puppet Master
- **Manifest**: a file that contains a set of instructions to be executed
- **Resource**: a portion of code that declares an element of the system and how its state should be changed. For instance, to install a package we need to define a *package* resource and ensure its state is set to "installed"
- **Module**: a collection of manifests and other related files organized in a pre-defined way to facilitate sharing and reusing parts of a provisioning
- **Class**: just like with regular programming languages, classes are used in Puppet to better organize the provisioning and make it easier to reuse portions of the code
- **Facts**: global variables containing information about the system, like network interfaces and operating system
- **Services**: used to trigger service status changes, like restarting or stopping a service

Puppet provisionings are written using a custom DSL (domain specific language) that is based on Ruby.

Resources

With Puppet, tasks or steps are defined by declaring **resources**. Resources can represent packages, files, services, users, and commands. They might have a state, which will trigger a system change in case the state of a declared resource is different from what is currently on the system. For instance, a *package* resource set to `installed` in your manifest will trigger a package installation on the system if the package was not previously installed.

This is what a *package* resource looks like:

```
package { 'nginx':
    ensure  => 'installed'
}
```

You can execute any arbitrary command by declaring an `exec` resource, like the following:

```
exec { 'apt-get update':
    command => '/usr/bin/apt-get update'
}
```

Note that the `apt-get update` portion on the first line is not the actual command declaration, but an identifier for this unique resource. Often we need to reference other resources from within a resource, and we use their identifier for that. In this case, the identifier is `apt-get update`, but it could be any other string.

Resource Dependency

When writing manifests, it is important to keep in mind that Puppet doesn't evaluate the resources in the same order they are defined. This is a common source of confusion for those who are getting started with Puppet. Resources must explicitly define dependency between each other, otherwise there's no guarantee of which resource will be evaluated, and consequently executed, first.

As a simple example, let's say you want execute a command, but you need to make sure a dependency is installed first:

```
package { 'python-software-properties':
    ensure => 'installed'
}

exec { 'add-repository':
    command => '/usr/bin/add-apt-repository ppa:ondrej/php5 -y'
    require => Package['python-software-properties']
}
```

The `require` option receives as parameter a reference to another resource. In this case, we are referring to the *Package* resource identified as `python-software-properties`. It's important to notice that while we use `exec`, `package`, and such for declaring resources (with lowercase), when referring to previously defined resources, we use `Exec`, `Package`, and so on (capitalized).

Now let's say you need to make sure a task is executed**before** another. For a case like this, we can use the `before` option instead:

```
package { 'curl':
    ensure => 'installed'
    before => Exec['install script']
}

exec { 'install script':
    command => '/usr/bin/curl http://example.com/some-script.sh'
```

Manifest Format

Manifests are basically a collection of resource declarations, using the extension `.pp`. Below you can find an example of a simple playbook that performs two tasks: updates the `apt` cache and installs `vim` afterwards:

```
exec { 'apt-get update':
    command => '/usr/bin/apt-get update'
}

package { 'vim':
    ensure => 'installed'
    require => Exec['apt-get update']
}
```

Before the end of this guide we will see a more real-life example of a manifest, explained in detail. The next section will give you an overview of the most important elements and features that can be used to write Puppet manifests.

Writing Manifests

Working with Variables

Variables can be defined at any point in a manifest. The most common types of variables are strings and arrays of strings, but other types are also supported, such as booleans and hashes.

The example below defines a string variable that is later used inside a resource:

```
$package = "vim"

package { $package:
   ensure => "installed"
}
```

Using Loops

Loops are typically used to repeat a task using different input values. For instance, instead of creating 10 tasks for installing 10 different packages, you can create a single task and use a loop to repeat the task with all the different packages you want to install.

The simplest way to repeat a task with different values in Puppet is by using arrays, like in the example below:

```
$packages = ['vim', 'git', 'curl']

package { $packages:
   ensure => "installed"
}
```

As of version 4, Puppet supports additional ways for iterating through tasks. The example below does the same thing as the previous example, but this time using the `each` iterator. This option gives you more flexibility for looping through resource definitions:

```
$packages.each |String $package| {
  package { $package:
    ensure => "installed"
  }
}
```

Using Conditionals

Conditionals can be used to dynamically decide whether or not a block of code should be executed, based on a variable or an output from a command, for instance.

Puppet supports most of the conditional structures you can find with traditional programming languages, like `if/else` and `case` statements. Additionally, some resources like `exec` will support attributes that work like a conditional, but only accept a command output as condition.

Let's say you want to execute a command based on a *fact*. In this case, as you want to test the value of a variable, you need to use one of the conditional structures supported, like `if/else`:

```
if $osfamily != 'Debian' {
 warning('This manifest is not supported on this OS.')
}
else {
 notify { 'Good to go!': }
}
```

Another common situation is when you want to condition the execution of a command based on the output from another command. For cases like this you can use `onlyif` or `unless`, like in the example below. This command will only be executed when the output from `/bin/which php` is successful, that is, the command exits with status **0**:

```
exec { "Test":
 command => "/bin/echo PHP is installed here > /tmp/test.txt",
 onlyif => "/bin/which php"
}
```

Similarly, `unless` will execute the command all times, except when the command under `unless` exits successfully:

```
exec { "Test":
 command => "/bin/echo PHP is NOT installed here > /tmp/test.txt",
 unless => "/bin/which php"
}
```

Working with Templates

Templates are typically used to set up configuration files, allowing for the use of variables and other features intended to make these files more versatile and reusable. Puppet supports two different formats for templates: Embedded Puppet (EPP) and Embedded Ruby (ERB). The EPP format, however, works only with recent versions of Puppet (starting from version 4.0).

Below is an example of an ERB template for setting up an Apache virtual host, using a variable for setting up the document root for this host:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot <%= @doc_root %>

    <Directory <%= @doc_root %>>
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

In order to apply the template, we need to create a `file` resource that renders the template content with the `template` method. This is how you would apply this template to replace the default Apache virtual host:

```
file { "/etc/apache2/sites-available/000-default.conf":
    ensure => "present",
    content => template("apache/vhost.erb")
}
```

Puppet makes a few assumptions when dealing with local files, in order to enforce organization and modularity. In this case, Puppet would look for a `vhost.erb` template file inside a folder `apache/templates`, inside your modules directory.

Defining and Triggering Services

Service resources are used to make sure services are initialized and enabled. They are also used to trigger service restarts.

Let's take into consideration our previous template usage example, where we set up an Apache virtual host. If you want to make sure Apache is restarted after a virtual host change, you first need to create a *service* resource for the Apache service. This is how such resource is defined in Puppet:

```
service { 'apache2':
    ensure => running,
    enable => true
}
```

Now, when defining the resource, you need to include a `notify` option in order to trigger a restart:

```
file { "/etc/apache2/sites-available/000-default.conf":
    ensure => "present",
    content => template("vhost.erb"),
    notify => Service['apache2']
}
```

Example Manifest

Now let's have a look at a manifest that will automate the installation of an Apache web server within an Ubuntu 14.04 system, as discussed in this guide's introduction.

The complete example, including the template file for setting up Apache and an HTML file to be served by the web server, can be found on Github. The folder also contains a Vagrantfile that lets you test the manifest in a simplified setup, using a virtual machine managed by Vagrant.

Below you can find the complete manifest:

default.pp

- `$doc_root = "/var/www/example"`
- `exec { 'apt-get update':`
- ` command => '/usr/bin/apt-get update'`
- `}`
- `package { 'apache2':`
- ` ensure  => "installed",`
- ` require => Exec['apt-get update']`
- `}`
- `file { $doc_root:`
- ` ensure => "directory",`
- ` owner => "www-data",`
- ` group => "www-data",`
- ` mode => 644`
- `}`
- `file { "$doc_root/index.html":`
- `   ensure => "present",`
- `   source => "puppet:///modules/main/index.html",`
- `   require => File[$doc_root]`
- `}`
- `file { "/etc/apache2/sites-available/000-default.conf":`
- `   ensure => "present",`
- `   content => template("main/vhost.erb"),`
- `   notify => Service['apache2'],`
- `   require => Package['apache2']`
- `}`
- `service { 'apache2':`
- `   ensure => running,`
- `   enable => true`
- `}`

Manifest Explained

line 1

The manifest starts with a variable definition, `$doc_root`. This variable is later used in a resource declaration.

lines 3-5

This **exec** resource executes an `apt-get update` command.

lines 7-10

This **package** resource installs the package `apache2`, defining that the `apt-get update` resource is a requirement, which means that it will only be executed after the required resource is evaluated.

lines 12-17

We use a **file** resource here to create a new directory that will serve as our document root. The `file` resource can be used to create directories and files, and it's also used for applying templates and copying local files to the remote server. This task can be executed at any point of the provisioning, so we didn't need to set any `require` here.

lines 19-23

We use another **file** resource here, this time to copy our local **index.html** file to the document root inside the server. We use the `source` parameter to let Puppet know where to find the original file. This nomenclature is based on the way Puppet handles local files; if you have a look at the Github example repository, you will see how the directory structure should be created in order to let Puppet find this resource. The document root directory needs to be created prior to this resource execution, that's why we include a `require` option referencing the previous resource.

lines 25-30

A new **file** resource is used to apply the Apache template and notify the service for a restart. For this example, our provisioning is organized in a module called **main**, and that's why the template source is **main/vhost.erb**. We use a `require` statement to make sure the template resource only gets executed after the package `apache2` is installed, otherwise the directory structure used by Apache may not be present yet.

lines 32-35

Finally, the **service** resource declares the `apache2` service, which we notify for a restart from the resource that applies the virtual host template.

Conclusion

Puppet is a powerful configuration management tool that uses an expressive custom DSL for managing server resources and automate tasks. Its language offers advanced resources that can give extra flexibility to your provisioning setups; it is important to remember that resources are not evaluated in the same order they are defined, and for that reason you need to be careful when defining dependencies between resources in order to establish the right chain of execution.

.