Using Puppet and Fabric for Server Deployment

November 10, 2013

http://read.sprabery.com/2013/11/10/Using-Puppet-and-Fabric-for-Server-Deployment/

In the past, I've used fabric to help in automating the deployment of Django Apps. Additionally, I used it to install dependencies, upload Nginx config files, and restart my applications.

Even after doing this, I felt it easier to spin up servers, get them configured, and then save an image of them, just using fabric to upload application specific config files and code. The entire fabric process has just felt fragile; fabric is a great tool, but it is important to use the right tool for the job.

This post is about how I've started using Puppet for configuration management, outlines the benefit of this kind of system, and has a walk through on how to get started *without* a so called "Puppet Master".

The Problem

Fabric does a great job of automating certain tasks. I even used fabric to generate this post which does a number of things like make a proper YAML header, sets the date with the proper filename, etc., things that I don't want to remember that I need to do after migrating my blog nearly six months ago.

The issue arises when using fabric for managing a large number of interconnected server side dependencies in a reusable manner. Fabric executes scripts, but I want my server to be in a certain *state*. And that's just the approach Puppet takes; it manages the *state* of servers.

Configuration Management

Defining a clear state for your servers, and letting the underlying software run whatever needs to be run to ensure that state is met instead of executing a series of scripts, is a much easier way of thinking of server management.

There is an added benefit of the modularity of Puppet scripts. For instance, I want all my Ubuntu instances to have UFW (Ubuntu Firewall) installed on them, along with Denyhosts, and a number of other personal tools I use such as vim and git. But if I'm running on bare metal (like on OVH), I may want those things, plus a specific ethernet configuration and the packages necessary for KVM based virtual machines. Puppet allows for fine grained modularity and stacked imports so that you can easily build classes of systems and reuse modules.

On top of that, you can store configuration files (those that typically go in /etc) with your Puppet "manifests" (a Puppet state definition to be applied to a server) and use a template language to fill in variables for specific server instances or groups of servers by providing the variables as arguments to manifest imports or generating them based on the kind of machine being used.

Getting Started

Ok, so you're sold, but with all the noise about configuration management, what is the best way to get started?

PuppetLabs (the makers of Puppet) has an excellent getting started guide and virtual machine download to get you up and going quickly. Spending some time with that can really make a difference, then just decide on something you want to do like Ubuntu + UFW + Denyhosts + Secure SSH Config for instance, and just get it working.

Admittedly, I didn't go through their guide step by step, instead I just choose a cloud provider like Digital Ocean and a goal (like the one mentioned above) and just did it. Digital Ocean is great for that because if you wipe out your networking, get those ssh keys wrong, or simply just want to start over, you can destroy and spin up a new machine effortlessly and likely spend under $1 even if you play with it for a week.

Typical Architecture

Typical Puppet deployments require that each of your servers be reachable by a domain name. You also have to set up a "Puppet Master" and install the Puppet daemon on your servers, which then will query the Puppet master to ensure that it meets the state requirements set for its domain name.

For larger organizations relying on spinning up many servers and using something like Amazon Route 53 for DNS management and EC2 for server creation, that may be great. But if you're just spinning up servers for side projects and want to get rolling quickly while still benefiting from configuration management, that may be over kill.

Puppet Solo (Fabric + Puppet)

My answer to this is using fabric to install the puppet module on the target server, rsync'ing my manifests folder, and then applying it to that server alone using puppet apply.

My folder structure looks like:

```
|-fabfile.py
|-requirements.txt
*|-puppet
  |---manifests
  |------kvm.pp
  |------vm.pp
    |---modules
      |-----denyhosts
        |-------files
        |----------denyhosts.conf
        |-------manifests
        |----------init.pp
      |-----kvm
        |-------files
        |----------interfaces
        |-------manifests
        |----------init.pp
      ...
      ...
      |-----nginx
        |-------files
        |----------nginx.conf
        |-------manifests
        |----------init.pp
```

As you can see, I have a fabfile in the root directory. I use this file to initialize new machines by installing the required ruby version and the puppet runtime (that converts manifest files into actions)

as a ruby gem. It then uploads the puppet subfolder and runs the ‘puppet apply’ command using either the kvm.pp file or the vm.pp, depending on the role of the machine. I use a modified version of the fabfile below.

```
from fabric.api import *

env.roledefs = {
  'vms': ['VM.IP.ADD.ONE', 'VM.IP.ADD.TWO'],
  'kvm': ['HOST.IP.ADD.RESS'],
}

def install_puppet(use_sudo=True):
  if use_sudo:
    sudo('apt-get -y install rsync ruby1.9.3 ruby1.9.1-full && gem install
puppet --no-ri --no-rdoc')
  else:
    run('apt-get -y install ruby1.9.3 ruby1.9.1-full && gem install puppet --no-
ri --no-rdoc')

def sync_puppet(use_sudo=True):
  if use_sudo:
    rsync_command = "rsync --rsync-path=\"sudo rsync\" --compress --recursive
--checksum \
    --delete  --itemize-changes ./puppet/ %s@%s:/var/puppet/" % (env.user,
env.host)
  else:
    rsync_command = "rsync --compress --recursive --checksum \
    --delete --itemize-changes ./puppet/ %s@%s:/var/puppet/"
    % (env.user, env.host)
  # run the rsync command
  local(rsync_command)

def apply(pp_file, use_sudo=True):
  with cd("/var/puppet"):
    if use_sudo:
      sudo("puppet apply --modulepath=./modules %s" % pp_file)
    else:
      run("puppet apply --modulepath=./modules %s" % pp_file)

@roles('kvm')
def initKvm():
  env.hosts = 'HOST.IP.ADD.RESS'
  env.user = 'USER'
  env.password = 'SET_ME'
  install_puppet(use_sudo=False)
  sync_puppet()
  apply(pp_file='manifests/kvm.pp', use_sudo=False)

@roles('kvm')
def updateKvm():
  env.host = 'HOST.IP.ADD.RESS'
  env.user = 'NEW_USER'
  env.password = ""
  sync_puppet()
  apply(pp_file='manifests/kvm.pp')

@roles('vms')
def update_vms():
  env.user = 'USER'
  env.password = ""
  sync_puppet()
  apply(pp_file='manifests/vms.pp')
```

The puppet subdirectory gets put into its own /var/puppet directory on the target server with `fab sync_puppet`. Fabric relies on the env.user, env.hosts, and env.password variables to be set before it can run, but will prompt for these if not provided, so any of the above functions can be run from the command line with `fab function_name`, and then you'll be prompted when additional info is needed.

I use rsync so that I can work on tuning scripts on a dummy vm before commiting them to version control; it also removes the dependency on an external git server for which you would have to manage credentials for on the target server.

The `apply` function above takes an argument for which top level manifest file to use and can be run with `fab apply:manifsts/vms.pp`. It will them prompt you for a host and apply the manifest file to that server.

I've used fabric roledefs to define groups of machines (such as all my vms) for which I want to apply a manifest file to. Now I can update my configuration across my machines with a single command (`fab update_vms`), no puppet master needed!

So lets look at an example of the top level manifest file vms.pp:

```
include ubuntu_base
include users
include sshd
include ufw
include denyhosts
```

Here, I'm installing a default set of packages defined in ubuntu_base/manifests/init.pp, creating users I have defined in users/manifests/init.pp, uploading sshd_config file stored in sshd/files/sshd_config, etc. In order to do that automatically with just `include MODULE`, we need to specify the module path in the `puppet apply` command. Looking back at the fabric script above, we can see that with:

```
puppet apply --modulepath=./modules manifests/vms.pp
```

As I continue, I can put application specific dependencies in a separate pp file in the root manifests directory, say web_app.pp that installs additional dependencies, and make a separate role in fabric for that machine if I need to. This way I can quickly and easily, and in a reusable manner, manage dependencies for varying machines, adding to machines exactly what they need. In my kvm.pp file, I have `include kvm.pp` that uploads network specific information, but other than that, it looks just like the vms.pp file.

Before I'm done, lets take a look at the denyhosts/manifests/init.pp file, as it highlights uploading config files, installing packages, and dependencies between rules (I want to install the package before changing the config file).

```
class denyhosts {
  package {'denyhosts':
    ensure=>latest
  }

  service {'denyhosts':
    ensure => running,
```

```
    enable => true,
    hasrestart => true,
    subscribe => File['denyhosts.conf']
  }

  file{"denyhosts.conf":
    path => "/etc/denyhosts.conf",
    owner => root,
    group => root,
    mode => 0640,
    require => Package['denyhosts'],
    source => "puppet:///modules/denyhosts/denyhosts.conf"
  }
}
```

The class name needs to match the folder structure under `modules`, as it allows you to import `denyhosts` from top level manifest files (like kvm.pp and vms.pp defined above).

The first structure of interest is the `package` keyword. It will ensure that the first string is installed and is the latest version in the distribution's package manager (but you can specify specific versions if you need to). One of the best things about this is that you don't have to worry which distro you use this with, it'll work with most popular distro tools including apt-get and yum.

Next, we have the `service` keyword. This ensures that the service is running, and then `enable` keyword ensures it is enabled at boot. The `subscribe` keyword makes the service reload on any changes to the file "denyhosts.conf". The `subscribe` keyword is a "metaparameter" that allows you to do ordering between resources (like 'file' in this example). You just take the other object identify (file), capitalize it, (File), and pass it the identifier ('denyhosts') that was used in its declaration. Check out the above link for a full list of the ordering metaparameters.

Last, we have the `file` resource. This takes a `source` metaparamter that defines where the file is located. There are various ways to define where the file resides. Here, I'm relying on puppet's internal file service, which typically queries the puppet master, but in this case `puppet:///modules/denyhosts/denyhosts.conf` is actually linking to `modules/denyhosts/files/denyhosts.conf`. This is an oddity, but it allows you to think of modules, the (module_name -> (manifests -> init.pp), (files -> file)) structure, as whole parts.

That basically covers it. It's easy to get started with Puppet, just use a similar directory structure, start playing with modules, and witness first hand the power of configuration management tools! If you have any questions or have a better way of doing things, I'd love to hear about it in the comments below!