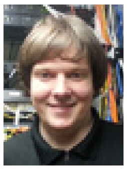
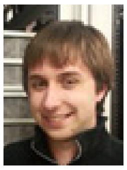
Deploying a Python App with Puppet

SPENCER KRUM AND WILLIAM VAN HEVELINGEN



Spencer Krum is a Linux and application administrator with UTI Worldwide, a shipping and logistics firm. He lives and works in Portland. He has been using Linux and Puppet for years. Krum is co-authoring *Pro Puppet, 2nd Edition* (http://www.apress.com/9781430260400), which should be available from Apress in October 2013. He is also writing an original book, *Beginning Puppet*, which should be available from Apress in late 2013. Krum helps maintain a number of public Puppet modules on the Puppet Forge. His favorite non-puppet open source project to commit to is the Ops School curriculum (opsschool.org), a project to build an Operations 101 handbook/manual for people who want to break into the operations engineering career field. He enjoys hacking, tennis, IRC bots, StarCraft, and Hawaiian food.
krum.spencer@gmail.com



William Van Hevelingen is the Unix Team Lead at the Computer Action Team (TheCAT), which provides IT support for the Maseeh College of Engineering and Computer Science at Portland State University. Van Hevelingen oversees the Linux/Unix systems and services for the college with the help of a small army of volunteer students. He is an active contributor to open source projects and is a co-author (with Spencer Krum and Ben Kero) of the second edition of *Pro Puppet*.
william.vanhevelingen@pdx.edu

In this article, we will explain how to deploy a simple Django app from source using Puppet [1]. Puppet is an open source configuration management tool developed by Puppet Labs, a Portland-based automation startup. The Puppet software pulls its configuration from code written in a Ruby DSL, which makes Puppet extremely configurable and pluggable. The application we are going to deploy is OSQA [2], an open source stack overflow-like web application. Because Puppet is distribution-agnostic, we can do this on any modern Linux. This recipe of Puppet code easily can be converted to your automatic deployment needs.

To deploy our web application, we are going to build a Puppet class and install some public modules. First make sure your system has git and Puppet 2.7.x or later installed. Puppet is available in the standard Ubuntu repositories as well as EPEL for Red Hat 6-based distributions. If you want the latest version of Puppet, which won't be required today, you can add the Puppet Labs package repository for your operating system. You will also need to use the Puppet Labs package repos if you are on a Red Hat 5-based distribution. You can also install Puppet from RubyGems.

Lets create a module to hold our class. We can use the Puppet utility to build the skeleton of the Puppet module:

```
$ puppet module generate demouser/osqa
Notice: Generating module at /root/demouser-osqa
demouser-osqa
demouser-osqa/spec
demouser-osqa/spec/spec_helper.rb
demouser-osqa/Modulefile
demouser-osqa/README
demouser-osqa/manifests
demouser-osqa/manifests/init.pp
demouser-osqa/tests
demouser-osqa/tests/init.pp

$ mv demouser-osqa/ /etc/puppet/modules/osqa
```

The vast majority of our code is going to be written into osqa/manifests/init.pp. We also need to pull in some public Puppet modules we will use for component tasks:

```
$ puppet module install puppetlabs/vcsrepo
$ puppet module install puppetlabs/apache
$ puppet module install puppetlabs/mysql
$ git clone https://github.com/stankevich/puppet-python /etc/puppet/modules/python
```

Deploying a Python App with Puppet

We're going to build the OSQA module part by part. If you want to cut to the chase and see the final version, you can look at https://github.com/nibalizer/puppet-module-osqa.

If you look in osqa/manifests/init.pp, you will find that the Puppet module tool has already created some boilerplate for you. You should come back later and fill out this documentation.

First, we need to add some parameters to this class so that it can be used by others:

```
class osqa (
 $install_dir     = '/home/osqa',
 $username        = 'osqa',
 $group           = 'osqa',
 $db_name         = 'osqa',
 $timezone        = 'America/Los_Angeles',
 $app_url         = 'http://puppet-article-4',
 $db_username     = 'osqa',
 $db_password     = 'changme!',
) {
…
```

This syntax means the class can be called with any of these parameters, but if any are omitted the the default on the right side will be used. Generally users will want to run this application as the OSQA user, and out of the /home/osqa directory, but someone might want to run it out of /var/www or /srv/www to be more congruent with their existing infrastructure.

Next we will create the user, group, and do other preliminary setup:

```
 group { $group:
   ensure => present,
 }

user { $username:
   ensure         => present,
   gid            => $group,
   managehome     => true,
   require        => Group[$username],
 }

 file { $install_dir:
   owner          => $username,
   recurse        => true,
   require        => Group[$username],
   before         => File["${install_dir}/requirements.txt"],
 }
```

These stanzas are Puppet resources. When the class is included on a host, these resources will be created. Notice that the user resource has a require => relationship with the group resource. Puppet is a declarative language; resources are not created in the order of the file, but in a random order. The way to break the randomness and chain logical dependencies is to use the require or before syntax.

Next we create resources for managing Apache. Because we are already including the Apache module, we can give very high-level directives here. Unfortunately, the Apache module is not really ready to manage WSGI applications, but we can work around that using the custom_fragment parameter and a file resource:

```
class { 'apache':
   default_vhost => false,
 }
 include apache::mod::wsgi

 # FIXME: 2013/08/16 apache module does not support wsgi yet
 file { '/etc/apache2/sites-enabled/wsgi.conf':
   ensure              => file,
   content             => "WSGISocketPrefix \${APACHE_RUN_DIR}
WSGI\nWSGIPythonHome ${install_dir}/virtenv-osqa",
   notify              => Service['apache2'],
 }

 # FIXME: 2013/08/16 apache module does not support wsgi yet
 apache::vhost { 'osqa-vhost':
   port                => 80,
   docroot             => "${install_dir}/osqa-server",
   custom_fragment     => "  WSGIDaemonProcess OSQA \n
WSGIProcessGroup OSQA\n  WSGIScriptAlias / ${install_dir}/
osqa-server/osqa.wsgi\n ",
   directories         => [
     { path => "${install_dir}/osqa-server/forum/upfiles", order
=> 'deny,allow', allow  => 'from all' },
     { path => "${install_dir}/osqa-server/forum/skins",   order
=> 'allow,deny', allow  => 'from all' }
   ],
   aliases             => [
     { alias => '/m/',       path =>
"${install_dir}/osqa-server/forum/skins/" },
     { alias => '/upfiles/' path =>
"${install_dir}/osqa-server/forum/upfiles/" }
   ],
   require             => Vcsrepo["${install_dir}/osqa-server"],
 }
```

Next we need a source checkout of our application. This particular application is using svn, but the vcsrepo resource below supports many version control systems, which is selected via the provider attribute:

```
vcsrepo { "${install_dir}/osqa-server":
  ensure    => present,
  provider  => svn,
  source    => 'http://svn.osqa.net/svnroot/osqa/trunk/',
  revision  => '1285',
  user      => $username,
  owner     => $group,
  require   => [User['osqa'], File[$install_dir]],
 }
```

After this we have to create some file resources and set some permissions that our application probably should create for itself, but Puppet can do just fine:

```
file { "${install_dir}/osqa-server/log":
  ensure    => directory,
  owner     => $username,
  group     => 'www-data',
  recurse   => true,
  mode      => '0775',
  require   => Vcsrepo["${install_dir}/osqa-server"],
 }

 file { "${install_dir}/osqa-server/log/django.osqa.log":
  owner     => $username,
  group     => 'www-data',
  mode      => '0664',
  require   => Vcsrepo["${install_dir}/osqa-server"],
 }

 $osqa_directories = [
  "${install_dir}/osqa-server/forum/upfiles",
  "${install_dir}/osqa-server/cache",
  "${install_dir}/cache",
  "${install_dir}/log",
  "${install_dir}/forum_modules"]

 file { $osqa_directories:
  ensure    => directory,
  group     => 'www-data',
  mode      => '0770',
  require   => Vcsrepo["${install_dir}/osqa-server"],
 }

 file { "${install_dir}/osqa-server":
  owner     => $username,
  group     => $group,
  recurse   => true,
  require   => Vcsrepo["${install_dir}/osqa-server"],
 }
```

Next we use Puppet's templating engine, which is the same ERB templating you've possibly been exposed to in Ruby web development, to create the wsgi file and configuration files for our application:

```
file { "${install_dir}/osqa-server/osqa.wsgi":
  content       => template('osqa/osqa.wsgi.erb'),
  require       => User['osqa'],
 }

 file { "${install_dir}/osqa-server/settings_local.py":
  owner         => $username,
  content       => template('osqa/settings_local.py.erb'),
  require       => Vcsrepo["${install_dir}/osqa-server"]
 }

 file { "${install_dir}/requirements.txt":
  content       => template('osqa/requirements.txt'),
  require       => Vcsrepo["${install_dir}/osqa-server"]
 }
```

We're templating out "requirements.txt" because the application doesn't ship with one. This further demonstrates how Puppet can be an effective deployment tool even in less than ideal circumstances.

The template files are stored as osqa/templates/filename.erb. You can check out the git repository for puppet-module-osqa if you would like to see them. (More information is available on ERB templating is available online at the Puppet Labs website and elsewhere.)

Next we will install and configure the MySQL server. Thanks to the MySQL module, this is painless:

```
class { 'mysql::server':
  config_hash => { 'root_password' => hiera('mysql_root_
password', 'changme!') },
 }

 package { 'libmysqlclient-dev':
  ensure => present,
 }

 include mysql::bindings::python

 mysql::db { $db_name:
  user        => $db_username,
  password    => $db_password,
  grant       => ['all'],
 }
```

Above we have used the *hiera* function call. Hiera allows us to look up data, like a database password above, in an external datastore. Commonly this datastore is just yaml files. This is useful because it allows us to separate data from code. Next we will install the Python virtual environment and install all the dependencies using pip inside that virtualenv. This is quick, easy, and simple thanks to the Python module:

```
class { 'python':
  version        => 'system',
  dev            => true,
  virtualenv     => true,
 }

 python::virtualenv { "${install_dir}/virtenv-osqa":
  ensure         => present,
  version        => 'system',
  systempkgs     => false,
  distribute     => true,
  requirements   => "${install_dir}/requirements.txt",
  owner          => $username,
  require        => [Vcsrepo["${install_dir}/osqa-server"],
Class['python'], File["${install_dir}/requirements.txt"]],
  notify         => Exec['syncdb'],
 }
```

The last set of resources are what Puppet calls "exec" resources. In any LAMP stack deployment, commands must be run for the application to configure the database. Puppet has the exec resource available to run any piece of shell the system administrator or developer wants to. Entering the virtual environment and running Django's manage.py is simple. The refreshonly directive coupled with the notify coming from the virtualenv means that these execs will only run right after the virtualenv is created, which will only happen on initial configuration, not continuously:

```
exec { 'syncdb':
  cwd            => "${install_dir}/osqa-server",
  provider       => shell,
  user           => $username,
  command        => ". ../virtenv-osqa/bin/activate && yes no |
${install_dir}/virtenv-osqa/bin/python manage.py syncdb --all",
  refreshonly    => true,
  notify         => Exec['migrate-forum'],
 }

 exec { 'migrate-forum':
  cwd            => "${install_dir}/osqa-server",
  provider       => shell,
  user           => $username,
  command        => ". ../virtenv-osqa/bin/activate &&
${install_dir}/virtenv-osqa/bin/python manage.py migrate forum
--fake",
  refreshonly => true,
```

With all our resources in place, we need to use another piece of Puppet syntax to chain them together in the correct way:

```
Class['python'] -> Python::Virtualenv <| |>
-> Python::Pip <| |> -> Class['mysql::server']
-> Mysql::Db[$db_name]
```

This syntax ensures that the Python class comes first, followed by its virtual environment and any pip resources, then the mysql::server class comes, followed by its MySQL database. When we try to run manage.py, we are required to have a database online.

With all that code entered, we can run this against a server with

```
$ puppet apply -e 'class { "osqa": } '
```

which will run for a while, then we have a functional OSQA installation up and running under mod_wsgi.

You can also use any of the parameters we allowed for above with the following syntax:

```
$ puppet apply -e 'class { "osqa": user => "web-osqa" }'
```

Or, if your environment already has puppet set up in master/agent mode, you could just add these class resources to the osqa server's node definition.

With that, we have built a simple Puppet module to deploy a Django web application. We are managing all of the primary components of the application: database, source code, Apache configuration, and virtual environment. We are also leveraging Puppet to overcome some of the limitations of the software, such as creating var and cache directories because the application doesn't create them itself. Puppet modules like this one can be used to streamline production deployment or to shorten iterative cycles in development.

References

[1] Puppet: docs.puppetlabs.com

[2] OSQA: http://www.osqa.net/download/