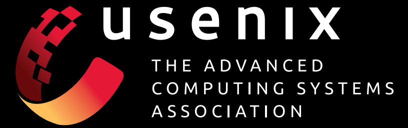
;login: logout

EXCLUSIVE ELECTRONIC EDITION

SEPTEMBER 2013



;login: logout

EXCLUSIVE ELECTRONIC EDITION SEPTEMBER 2013

2 Sysadmin Tools for Tackling the Cloud
Mark Hinkle

5 Deploying a Python App with Puppet
Spencer Krum and William Van Hevelingen

9 Configuring Your Linux System with the CFEngine Design Center
Diego Zamboni

14 The Slow Winter
James Mickens

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Michele Nelson

USENIX ASSOCIATION
2560 Ninth Street, Suite 215,
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738
www.usenix.org

©2013 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

Sysadmin Tools for Tackling the Cloud

MARK HINKLE

Mark Hinkle is the Senior Director, Open Source Solutions, at Citrix. He joined Citrix as a result of their July 2011 acquisition of Cloud.com, where he was Vice President of Community. He is currently responsible for Citrix Open Source Business Office and the Citrix efforts around Apache CloudStack, Open Daylight, Xen Project, and XenServer. Previously, Hinkle was the VP of Community at Zenoss Inc., a producer of the open source application, server, and network management software. He also is a longtime open source expert and author, having served as editor in chief for both *LinuxWorld Magazine* and *Enterprise Open Source Magazine*. Mark Hinkle wrote the book *Windows to Linux Business Desktop Migration* (Thomson, 2006). He is a contributor to NetworkWorld's Open Source Subnet, and his personal blog on open source, technology, and new media can be found at www.socializedsoftware.com. Follow him on Twitter @mrhinkle. mrhinkle@socializesoftware.com

If you read much in the way of the tech media you are probably numb from the cloud computing hype. You have real problems servers to run, backups to execute, and networks to configure. These things exist in two worlds for most sysadmins: your data center and an increasing number of third-party cloud services. Adding the following tools to your sysadmin toolbox will allow you to take advantage of the cloud without missing a beat.

Command and Control Stacks

Most system administrators who do things at scale are already utilizing configuration management tools such as Cfengine, Puppet and/or Chef. All three have considerable capabilities for automation and configuration; however as you move to the cloud, considering tools that utilize access methods that will likely exist natively on your cloud infrastructure as well as your legacy metal is prudent.

Ansible [1] is a simple open source orchestration stack that allows you to communicate with servers over SSH. In turn, this allows you to communicate with machines via a protocol that's likely already available on your machine to execute commands over SSH defined in YAML to call programs written in virtually any language, including Python, Ruby, or even Perl.

SaltStack [2] is another open source configuration management and execution framework along the same lines as Ansible, but it differs by using ZeroMQ as a message bus to execute changes across a network in parallel.

Cloud Controllers

You likely already have a bunch of infrastructure running on your metal and, as that infrastructure goes out of service, you probably are looking to move some of these workloads to the cloud. The decision you may struggle with is determining which cloud. Until you move those services, you won't know how your applications perform and the nuances of each cloud. If you are of a devops mentality, you probably have or will design systems that are easily replicated infrastructure across different architectures, cloud or otherwise. These tools will help you achieve that goal.

If you made your choice already, you may be instrumenting your shop to a certain API. Otherwise, one strategy is to instrument to a single cloud controller that has API compatibility with multiple clouds. Jclouds [3], an Apache incubator project (however, it is very mature), is a perfect example of this type of technology. Jclouds is a library that furnishes a single source to develop tools against—but still broker calls to—multiple clouds through the use of portable cloud abstractions. Jclouds users can take advantage of Java or Clojure as the domain-specific language (DSL). Python experts can take advantage of similar functionality in Apache Libcloud [4], and Ruby enthusiasts can take advantage of their existing skills using Fog [5].

Sysadmin Tools for Tackling the Cloud

Storage

As you start to utilize cloud services, you will quickly realize the advantages and challenges of deploying infrastructure in a much more geographically diverse landscape. Often the challenge is to provide data in a distributed environment with varying levels of utility. For example, you may have data that needs to be stored with varying degrees of availability and integrity. Google has done a considerable amount of research in this area [6] and, if you want to geek out on the considerations for globally distributed data, their findings are informative. Whereas Google focuses on the why, I would direct you to the how.

Gluster [7] is a network/cluster file system written in user space and uses Filesystem in User Space (FUSE) to hook itself with virtual file system (VFS) layer. Gluster works with common concrete file systems, such as ext3, ext4, and xfs. In terms of random file access, the more servers you add the better this scales. Common use cases are for content that is replicated and served behind caching services, such as images or music files delivered by Pandora, a rumored Gluster user.

Ceph [8] is similar in that it is an open source file system and distributed file store that can provide storage, much like Amazon's S3 or Block storage, through their RADOS block device for KVM and additional hypervisors, which will be added soon.

Built on top of Basho's Riak NoSQL database, Riak CS [9] is yet another open source object store. Riak provides a highly available, fault-tolerant storage system that includes compatibility with Amazon's S3 API. Riak CS can provide storage for images, documents, VM backups, archives of information, and other large objects on utility hardware, providing a foundation for storage that compliments your cloud at a much more attractive price point than legacy enterprise storage solutions.

Summary

There are lots of reasons to move to the cloud, and plenty of reasons to continue managing your own infrastructure. The solutions described in this article augment either strategy by providing tools to help you automate your increasingly distributed infrastructure, which lets you keep your options open as you explore new cloud services or look for affordable storage solutions that are uniquely suited to the cloud.

References

[1] Ansible: http://www.ansibleworks.com/

[2] SaltStack: http://saltstack.com/community.html

[3] Jclouds: http://jclouds.incubator.apache.org/

[4] Apache Libcloud: http://libcloud.apache.org/

[5] Fog: http://fog.io/

[6] Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V., Barroso, L., Grimes, C., and Quinlan, S., 2010, Availability in globally distributed storage systems, Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, USENIX (2010), http://www.usenix.org/events/osdi10/tech/full_papers/Ford.pdf

[7] Gluster: http://www.glusterfs.org/

[8] Ceph: http://ceph.com/

[9] Riak CS: http://basho.com/riak-cloud-storage/

OPEN ACCESS

Why Join USENIX?

We support members' professional and technical development through many ongoing activities, including:

- Open access to research presente d at our events
- Workshops on hot topics
- Conferences presenting the latest in research and practice
- LISA: The USENIX Special Interest Group for Sysadmins
- *;login:*, the magazine of USENIX
- Student outreach

Your membership dollars go towards programs including:

- Open access policy: All conference papers and videos are immediately free to everyone upon publication
- Student program, including grants for conference attendance
- Good Works program

Helping our many communities share, develop, and adopt ground-breaking ideas in advanced technology

Join us at www.usenix.org

;login: Limited Time *;login:* Subscription Offer Now Available

Enjoying the free article? Take advantage of this special offer: For only $55, get a 12 month subscription to the electronic edition of *;login:*, the highly regarded bimonthly USENIX magazine, as well as access to *;login: logout*, our new exclusive electronic publication. *;login:* is available in PDF, ePub, and Mobi formats.

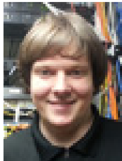
;login:, offers a selection of articles, conference reports, book reviews, and research, which strive to present the most exciting and relevant information to our community. Themed issues focus on system administration, file systems, security, networking, operating systems, and more. A sampling of previous articles include:

- "Ganeti: Cluster Virtualization Manager," *Trotter and Limoncelli, June 2013*
- "A Study of Linux File System Evolution," *Lu et al., June 2013*
- "Interview with Ted Ts'o," *Farrow, June 2013*
- "Do Users Verify SSH Keys?," *Gutmann, August 2011*
- "For Extreme Parallelism, Your OS Is Sooooo Last-Millennium," *Knauerhase, Cledat, and Teller, October 2012*

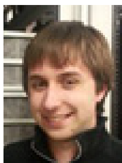
To subscribe, please go to www.usenix.org/login/promotion and enter coupon code LOGINSUB10 at checkout.

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 boiler-plate 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

Deploying a Python App with Puppet

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/

Configuring Your Linux System with the CFEngine Design Center

DIEGO ZAMBONI

Diego Zamboni is a computer scientist, consultant, author, programmer, sysadmin, and overall geek who works as a senior security advisor at CFEngine. He has more than 20 years of experience in system administration and security, and has worked in both the applied and theoretical sides of the computer science field. He holds a Ph.D. from Purdue University, and has worked as a sysadmin at a supercomputer center, as a researcher at the IBM Zurich Research Lab, and as a consultant at HP Enterprise Services. Zamboni is the author of the book "Learning CFEngine 3", published by O'Reilly Media. He lives in Queretaro, Mexico with his wife and two daughters. diego.zamboni@cfengine.com

CFEngine is an efficient, lightweight, and powerful configuration management tool for computer systems of all kinds. The most recent version, CFEngine 3.5.2, was released in August 2013. With CFEngine, you can express the desired state of your systems in two main ways: by writing policy in the CFEngine policy language directly, or by using the CFEngine Design Center [1], a repository of ready-to-use components called sketches, which allow you to perform entirely data-driven configuration. There are sketches for all sorts of tasks, from basic system configuration to complex cloud deployments. In this article, I will use simple examples to show you how to perform basic configuration tasks using the Design Center.

Getting Ready

First, you must install CFEngine and the Design Center on your system. Two versions of CFEngine are available: the open-source version (CFEngine Community) and the commercial version (CFEngine Enterprise). In my examples, I will use the Community version, which is available as packages for most Linux distributions and can also be downloaded and compiled from source code [2].

The easiest way to set up a test environment is to use Vagrant [3]. If you have Vagrant installed, fetch the sample Vagrantfile [4], put it in a directory, run *vagrant up*, and you will have a freshly installed Ubuntu 12.04 VM with both CFEngine and the Design Center ready to use. Then you can skip the rest of this section. If you prefer to do this on your own machine, or you don't want to use Vagrant, follow the instructions below.

I will use a fresh Ubuntu 12.04/64bit install, and follow the instructions from https://cfengine.com/cfengine-linux-distros to install CFEngine (command output edited for brevity):

```
# wget -q http://cfengine.com/pub/gpg.key
# apt-key add gpg.key
# rm gpg.key
# echo "deb http://cfengine.com/pub/apt $(lsb_release -cs) main" >  \
> /etc/apt/sources.list.d/cfengine-community.list
# apt-get -qq update
# apt-get -qq install cfengine-community
Selecting previously unselected package cfengine-community.
...
```

Now CFEngine is installed but not running. For this, we need to bootstrap CFEngine to a policy server. We will set up our machine as its own policy server, so we need to bootstrap to its own IP address:

Configuring Your Linux System with the CFEngine Design Center

```
# ifconfig eth0
eth0     Link encap:Ethernet  HWaddr 08:00:27:fe:aa:af
         inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
...
# /var/cfengine/bin/cf-agent --bootstrap 10.0.2.15
2013-08-19T22:25:53+0000   notice: Q: "...f-serverd"": 2013-08-
19T22:25:53+0000   notice: Server is starting...
2013-08-19T22:25:53+0000   notice: R: This host assumes the
role of policy server
2013-08-19T22:25:53+0000   notice: R: Updated local policy from
policy server
2013-08-19T22:25:53+0000   notice: R: Started the server
2013-08-19T22:25:53+0000   notice: R: Started the scheduler
2013-08-19T22:25:53+0000   notice: Bootstrap to '10.0.2.15'
completed successfully!
```

Now CFEngine is running, which you can verify by looking at the running processes:

```
# ps  ax | grep cf-
 1869 ?        Ss     0:00 /var/cfengine/bin/cf-execd
 1875 ?        Ss     0:00 /var/cfengine/bin/cf-serverd
 1889 ?        Ss     0:00 /var/cfengine/bin/cf-monitord
```

From now on, the CFEngine command *cf-agent* will run every five minutes to execute its policies. In this article, I will not go into more detail about how CFEngine works, but rather show you how you can use the CFEngine Design Center to configure your system without having to write CFEngine policies. The Design Center is hosted on GitHub [5], and its repository includes both the sketches and the tools used to manage them. We will clone the repository using *git*:

```
# apt-get -qq install git libterm-readline-gnu-perl
# cd /var/cfengine/
# git clone https://github.com/cfengine/design-center
Cloning into 'design-center'...
...
```

Using the CFEngine Design Center

Now we are ready to start using the Design Center. From the command line, the *cf-sketch* tool is the main way to manage Design Center sketches on your systems. CFEngine Enterprise includes a GUI for the Design Center, but for now we will stick to the command-line tools.

First, we need to run *cf-sketch*, which will put us in an interactive prompt:

```
# cd /var/cfengine/design-center/tools/cf-sketch/
# ./cf-sketch.pl
Welcome to cf-sketch version 3.5.0b1.
CFEngine AS, 2013.

Enter any command to cf-sketch, use 'help' for help, or 'quit' or
'^D' to quit.

cf-sketch> _
```

You can type *help* at this prompt to see all the commands available. In particular, you can type *search* to produce a listing of all the sketches available in the repository. For now, we will dive straight into the configuration of our system.

Let's look at some of the system configuration sketches available in the Design Center:

```
cf-sketch> search system

The following sketches match your query:

System::Logrotate Manage log rotation settings
System::Routes Manage system routes
System::Sudoers Sets defaults and user permissions in the
sudoers fileSystem::Syslog Configures syslog
System::access Manage access.conf values
System::config_resolver Configure DNS resolver
System::cron Manage crontab and /etc/cron.d contents
System::etc_hosts Manage /etc/hosts
System::motd Configure the Message of the Day
System::set_hostname Set system hostname. Domain name is also
set on Mac, Red Hat and and Gentoo derived distributions (but
not Debian).
System::sysctl Manage sysctl values
System::tzconfig Manage system timezone configuration
```

First we will configure the system timezone. For this, we will use the *System::tzconfig* sketch. We can use the *info* command to get detailed information about the sketch, including the parameters it uses:

```
cf-sketch> info -v System::tzconfig

The following sketches match your query:

Sketch System::tzconfig
Description: Manage system timezone configuration
Authors: Nick Anderson <nick@cmdln.org>, Ted Zlatanov <tzz@
lifelogs.com>
Version: 1.2
License: MIT
Tags: cfdc
Installed: No
Parameters:
  For bundle set
    timezone: string
    zoneinfo: string
Return values:
```

```
  Bundle set: [ timezone ]
```

The first step is to install it:

```
cf-sketch> install System::tzconfig

Sketch System::tzconfig installed under /var/cfengine/
masterfiles/sketches.
```

We can verify that the sketch has been installed using the *list* command. Note that a couple of library sketches were automatically installed as dependencies of *System::tzconfig*:

```
cf-sketch> list

The following sketches are installed:

CFEngine::dclib Design Center standard library
CFEngine::stdlib The portions of the CFEngine standard library
(also known as COPBL) that are compatible with 3.4.0 releases
System::tzconfig Manage system timezone configuration
```

Next, we need to define a parameter set for our sketch, which contains the values of the parameters needed by the sketch (enter your own timezone instead of the one shown here):

```
cf-sketch> define params System::tzconfig

Please enter a name for the new parameter set (default:
System::tzconfig-set-000): tzconfig1
Querying configuration for parameter set 'tzconfig1' for bundle
'set'.
Please enter parameter timezone.
  (enter STOP to cancel)
timezone : Mexico/General
Please enter parameter zoneinfo.
  (enter STOP to cancel)
zoneinfo : /usr/share/zoneinfo
Defining parameter set 'tzconfig1' with the entered data.
Parameter set tzconfig1 successfully defined.
```

Now we need to define an environment, which is short for "set of conditions under which a sketch will be executed with certain parameters". The conditions are expressed as CFEngine class expressions, so they can represent arbitrary conditions on the system, either automatically detected by CFEngine or set by your own CFEngine policies. For our example, we will activate our sketches in all Linux machines, so we will use the *linux* class, which is automatically set by CFEngine when it runs on a Linux host:

```
cf-sketch> define env

Please enter a name for the new environment: env_linux
I will now prompt you for the conditions for activation, test,
and verbose mode
that will be associated with environment 'env_linux'. Please
enter them as
CFEngine class expressions.
Please enter the activation condition: linux
Please enter the test condition: !any
Please enter the verbose condition: !any
Environment 'env_linux' successfully defined.
```

Now all we need to do is activate our sketch, telling it that we want to run it with the parameter set we defined, on all Linux machines:

```
cf-sketch> activate System::tzconfig tzconfig1 env_linux

Using generated activation ID 'System::tzconfig-1'.
Using existing parameter definition 'tzconfig1'.
Using existing environment 'env_linux'.
Activating sketch System::tzconfig with parameters tzconfig1.
```

Note that both parameter sets and environments have names, and all that the activate command does is "tie together" a sketch, a parameter set, and an environment.

After we have activated a sketch, we need to deploy and execute it, which can be done as a one-time operation using the run command (mostly for testing your parameters). Note the change in the system timezone before and after the sketch executes:

```
# date
Tue Aug 20 06:40:29 UTC 2013
#./cf-sketch.pl

cf-sketch> list activations

The following activations are defined:

Activation ID System::tzconfig-1
  Sketch: System::tzconfig
  Parameter sets: [ tzconfig1 ]
  Environment:  'env_linux'

cf-sketch> run

Runfile /var/cfengine/masterfiles/cf-sketch-runfile-standalone.
cf successfully generated.
Now executing the runfile with: /usr/local/sbin/cf-agent  -f /
var/cfengine/masterfiles/cf-sketch-runfile-standalone.cf

2013-08-20T06:40:47+0000   notice: R: System timezone updated
to Mexico/General
```

Configuring Your Linux System with the CFEngine Design Center

```
cf-sketch>
# date
Tue Aug 20 01:40:51 CDT 2013
```

Of course, you don't want to run the sketches manually, when the purpose of CFEngine is to keep your systems automatically configured. To automate the process, we must incorporate the execution of the sketches into the periodic execution of CFEngine by using the deploy command:

```
cf-sketch> deploy

Runfile /var/cfengine/masterfiles/cf-sketch-runfile.cf
successfully generated.
```

In the current release of CFEngine, we must make one change to the included CFEngine policy files in order for the sketches to be properly loaded. Open the */var/cfengine/masterfiles/promises.cf* file and find this section:

```
        # COPBL/Custom libraries.  Eventually this should use
wildcards.
             @(cfengine_stdlib.inputs),

         # Design Center
             # MARKER FOR CF-SKETCH INPUT INSERTION
             "cf-sketch-runfile.cf",
```

Because sketches load their own libraries, we must comment out the line that loads the CFEngine standard library and add a line that loads sketch-required files. The end result looks like this:

```
        # COPBL/Custom libraries.  Eventually this should use
wildcards.
         #   @(cfengine_stdlib.inputs),

         # Design Center
             # MARKER FOR CF-SKETCH INPUT INSERTION
             "cf-sketch-runfile.cf",
             @(cfsketch_g.inputs),
```

Now the sketch we activated will be executed every five minutes to check whether anything needs to be fixed. If you manually change the timezone of your system, you will notice that within five minutes it will change back to the one you configured in the sketch.

We will now configure two additional sketches for basic system configuration tasks, following the same install-parameters-activate sequence we already saw. First, we will use CFEngine to maintain our */etc/motd* file:

```
cf-sketch> install System::motd

Sketch System::motd installed under /var/cfengine/masterfiles/
sketches.

cf-sketch> define params System::motd

Please enter a name for the new parameter set (default:
System::motd-entry-000): motd1
Querying configuration for parameter set 'motd1' for bundle
'entry'.
Please enter parameter motd (Message of the Day (aka motd)).
  (enter STOP to cancel)
motd : This sytem is managed by CFEngine. Go away!
Please enter parameter motd_path (Location of the primary,
often only, MotD file).
  (enter STOP to cancel)
motd_path [/etc/motd]: /etc/motd
Please enter parameter prepend_command (Command output to
prepend to MotD).
  (enter STOP to cancel)
prepend_command [/bin/uname -snrvm]: /bin/uname -snrvm
Please enter parameter dynamic_path (Location of the dynamic
part of the MotD file).
  (enter STOP to cancel)
dynamic_path :
Please enter parameter symlink_path (Location of the symlink to
the motd file).
  (enter STOP to cancel)
symlink_path :
Defining parameter set 'motd1' with the entered data.
Parameter set motd1 successfully defined.

cf-sketch> activate System::motd motd1 env_linux

Using generated activation ID 'System::motd-1'.
Using existing parameter definition 'motd1'.
Using existing environment 'env_linux'.
Activating sketch System::motd with parameters motd1.
```

Note that we do not need to define additional environments for these sketches; they are being activated using the same *env_linux* environment we prepared previously.

We will also use CFEngine to maintain some security-related parameters in the system's sshd configuration:

```
cf-sketch> install Security::SSH

Sketch Security::SSH installed under /var/cfengine/masterfiles/
sketches.

cf-sketch> define params Security::SSH
```

```
Please enter a name for the new parameter set (default:
Security::SSH-sshd-000): ssh1
Querying configuration for parameter set 'ssh1' for bundle
'sshd'.
Please enter parameter params.
  (enter STOP to cancel)
Next key (Enter to finish): PermitRootLogin
params[PermitRootLogin]: no
Next key (Enter to finish): X11Forwarding
params[X11Forwarding]: no
Next key (Enter to finish):
Defining parameter set 'ssh1' with the entered data.
Parameter set ssh1 successfully defined.

cf-sketch> activate Security::SSH ssh1 env_linux

Using generated activation ID 'Security::SSH-1'.
Using existing parameter definition 'ssh1'.
Using existing environment 'env_linux'.
Activating sketch Security::SSH with parameters ssh1.
```

Before those sketches take any effect, they must be deployed:

```
cf-sketch> deploy

Runfile /var/cfengine/masterfiles/cf-sketch-runfile.cf
successfully generated.
```

Within a few minutes, you should see those changes reflected in your system:

```
# cat /etc/motd
This sytem is managed by CFEngine. Go away!
# egrep 'PermitRoot|X11For' /etc/ssh/sshd_config
PermitRootLogin no
X11Forwarding no
```

We are done! Your system automatically will be maintained according to the criteria you set. Try modifying any of these settings to see how CFEngine automatically brings them back into compliance. To get a better idea of all the things you can do with the Design Center, I encourage you to explore the available sketches.

Conclusion

In this article, I have only touched on the surface of what the Design Center can do. To learn more about the Design Center's capabilities and how to contribute new sketches, read the CFEngine documentation at http://cfengine.com/docs.

References

[1] CFEngine Design Center: http://cfengine.com/cfengine-design-center/

[2] Download CFEngine: https://cfengine.com/downloads.

[3] Vagrant: http://www.vagrantup.com/

[4] Vagrantfile: https://raw.github.com/cfengine/vagrant-cfengine-provisioner/master/sample/community_vagrant-1.2/Vagrantfile

[5] CFEngine Design Center on GitHub: https://github.com/cfengine/design-center

The Slow Winter

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

According to my dad, flying in airplanes used to be fun. You could smoke on the plane, and smoking was actually good for you. Everybody was attractive, and there were no fees for anything, and there was so much legroom that you could orient your body parts in arbitrary and profane directions without bothering anyone, and you could eat caviar and manatee steak as you were showered with piles of money that were personally distributed by JFK and The Beach Boys. Times were good, assuming that you were a white man in the advertising business, WHICH MY FATHER WAS NOT SO PERHAPS I SHOULD ASK HIM SOME FOLLOW-UP QUESTIONS BUT I DIGRESS. The point is that flying in airplanes used to be fun, but now it resembles a dystopian bin-packing problem in which humans, carry-on luggage, and five dollar peanut bags compete for real estate while crying children materialize from the ether and make obscure demands in unintelligible, Wookie-like languages while you fantasize about who you won't be helping when the oxygen masks descend.

I think that it used to be fun to be a hardware architect. Anything that you invented would be amazing, and the laws of physics were actively trying to help you succeed. Your friend would say, "I wish that we could predict branches more accurately," and you'd think, "maybe we can leverage three bits of state per branch to implement a simple saturating counter," and you'd laugh and declare that such a stupid scheme would never work, but then you'd test it and it would be 94% accurate, and the branches would wake up the next morning and read their newspapers and the headlines would say OUR WORLD HAS BEEN SET ON FIRE. You'd give your buddy a high-five and go celebrate at the bar, and then you'd think, "I wonder if we can make branch predictors even more accurate," and the next day you'd start XOR'ing the branch's PC address with a shift register containing the branch's recent branching history, because in those days, you could XOR anything with anything and get something useful, and you test the new branch predictor, and now you're up to 96% accuracy, and the branches call you on the phone and say OK, WE GET IT, YOU DO NOT LIKE BRANCHES, but the phone call goes to your voicemail because you're too busy driving the speed boats and wearing the monocles that you purchased after your promotion at work. You go to work hung-over, and you realize that, during a drunken conference call, you told your boss that your processor has 32 registers when it only has 8, but then you realize THAT YOU CAN TOTALLY LIE ABOUT THE NUMBER OF PHYSICAL REGISTERS, and you invent a crazy hardware mapping scheme from virtual registers to physical ones, and at this point, you start seducing the spouses of the compiler team, because it's pretty clear that compilers are a thing of the past, and the next generation of processors will run English-level pseudocode directly. Of course, pride precedes the fall, and at some point, you realize that to implement aggressive out-of-order execution, you need to fit more transistors into the same die size, but then a material science guy pops out of a birthday

The Slow Winter

cake and says YEAH WE CAN DO THAT, and by now, you're touring with Aerosmith and throwing Matisse paintings from hotel room windows, because when you order two Matisse paintings from room service and you get three, that equation is going to be balanced. It all goes so well, and the party keeps getting better. When you retire in 2003, your face is wrinkled from all of the smiles, and even though you've been sued by several pedestrians who suddenly acquired rare paintings as hats, you go out on top, the master of your domain. You look at your son John, who just joined Intel, and you rest well at night, knowing that he can look forward to a pliant universe and an easy life.

Unfortunately for John, the branches made a pact with Satan and quantum mechanics during a midnight screening of "Weekend at Bernie's II." In exchange for their last remaining bits of entropy, the branches cast evil spells on future generations of processors. Those evil spells had names like "scaling-induced voltage leaks" and "increasing levels of waste heat" and "Pauly Shore, who is only loosely connected to computer architecture, but who will continue to produce a new movie every three years until he sublimates into an empty bag of Cheetos and a pair of those running shoes that have individual toes and that make you look like you received a foot transplant from a Hobbit, Sasquatch, or an infertile Hobbit/Sasquatch hybrid." Once again, I digress. The point is that the branches, those vanquished foes from long ago, would have the last laugh.

When John went to work in 2003, he had an indomitable spirit and a love for danger, reminding people of a less attractive Ernest Hemingway or an equivalently attractive Winston Churchill. As a child in 1977, John had met Gordon Moore; Gordon had pulled a quarter from behind John's ear and then proclaimed that he would pull twice as many quarters from John's ear every 18 months. Moore, of course, was an incorrigible liar and tormentor of youths, and he never pulled another quarter from John's ear again, having immediately fled the scene while yelling that Hong Kong will always be a British territory, and nobody will ever pay $8 for a Mocha Frappuccino, and a variety of other things that seemed like universal laws to people at the time, but were actually just arbitrary nouns and adjectives that Moore had scrawled on a napkin earlier that morning. Regardless, John was changed forever, and when he grew up and became a hardware architect, he poured all of his genius into making transistors smaller and more efficient. For a while, John's efforts were rewarded with ever-faster CPUs, but at a certain point, the transistors became so small that they started to misbehave. They randomly switched states; they leaked voltage; they fell prey to the seductive whims of cosmic rays that, unlike the cosmic rays in comic books, did not turn you into a superhero, but instead made your transistors unreliable and shiftless, like a surly teenager who is told to clean his room and who will occasionally just spray his bed with Lysol and declare victory.

As the transistors became increasingly unpredictable, the foundations of John's world began to crumble. So, John did what any reasonable person would do: he cloaked himself in a wall of denial and acted like nothing had happened. "Making processors faster is increasingly difficult," John thought, "but maybe people won't notice if I give them more processors." This, of course, was a variant of the notorious Zubotov Gambit, named after the Soviet-era car manufacturer who abandoned its attempts to make its cars not explode, and instead offered customers two Zubotovs for the price of one, under the assumption that having two occasionally combustible items will distract you from the fact that both items are still occasionally combustible. John quietly began to harness a similar strategy, telling his marketing team to deemphasize their processors' speed, and emphasize their level of parallelism.

At first, John's processors flew off the shelves. Indeed, who wouldn't want an octavo-core machine with 73 virtual hyper-threads per physical processor? Alan Greenspan's loose core policy and weak parallelism regulation were declared a resounding success, and John sipped on champagne as he watched the money roll in. However, a bubble is born so that a bubble can pop, and this one was no different. John's massive parallelism strategy assumed that lay people use their computers to simulate hurricanes, decode monkey genomes, and otherwise multiply vast, unfathomably dimensioned matrices in a desperate attempt to unlock eigenvectors whose desolate grandeur could only be imagined by Edgar Allen Poe.

Of course, lay people do not actually spend their time trying to invert massive hash values while rendering nine copies of the Avatar planet in 1080p. Lay people use their computers for precisely ten things, none of which involve massive computational parallelism, and seven of which involve procuring a vast menagerie of pornographic data and then curating that data using a variety of fairly obvious management techniques, like the creation of a folder called "Work Stuff," which contains an inner folder called "More Work Stuff," where "More Work Stuff" contains a series of ostensible documentaries that describe the economic interactions between people who don't have enough money to pay for pizza and people who aren't too bothered by that fact. Thus, when John said "imagine a world in which you're constantly executing millions of parallel tasks," it was equivalent to saying "imagine a world that you do not and will never live in." Indeed, a world in which you're constantly simulating nuclear explosions while rendering massive 3-D environments is a world that's been taken over by members of a high school A.V. club. The members of a high school A.V. club

The Slow Winter

lack the chops to establish a global dictatorship, if only because doing such a thing would require them to reduce their visits to Renaissance festivals, and those turkey legs need help to be consumed in the style of a 15th century Italian aristocrat.

John was terrified by the collapse of the parallelism bubble, and he quickly discarded his plans for a 743-core processor that was dubbed The Hydra of Destiny and whose abstract Platonic ideal was briefly the third-best chess player in Gary, Indiana. Clutching a bottle of whiskey in one hand and a shotgun in the other, John scoured the research literature for ideas that might save his dreams of infinite scaling. He discovered several papers that described software-assisted hardware recovery. The basic idea was simple: if hardware suffers more transient failures as it gets smaller, why not allow software to detect erroneous computations and re-execute them? This idea seemed promising until John realized THAT IT WAS THE WORST IDEA EVER. Modern software barely works when the hardware is correct, so relying on software to correct hardware errors is like asking Godzilla to prevent Mega-Godzilla from terrorizing Japan. THIS DOES NOT LEAD TO RISING PROPERTY VALUES IN TOKYO. It's better to stop scaling your transistors and avoid playing with monsters in the first place, instead of devising an elaborate series of monster checks-and-balances and then hoping that the monsters don't do what monsters are always going to do because if they didn't do those things, they'd be called dandelions or puppy hugs.

At this point, John was living under a bridge and wearing a bird's nest as a hat. Despite his tragic sartorial collaborations with the avian world, John still believed that somehow, some way, he could continue to make his transistors smaller. Perhaps the processor could run multiple copies of each program, comparing the results to detect errors? Perhaps a new video codec could tolerate persistently hateful levels of hardware error? All of these techniques could be implemented. However, John slowly realized that these solutions were just things that he could do, and inventing "a thing that you could do" is a low bar for human achievement. If I were walking past your house and I saw that it was on fire, I could try to put out the fire by finding a dingo and then teaching it how to speak Spanish. That's certainly a thing that I could do. However, when you arrived at your erstwhile house and found a pile of heirloom ashes, me, and a dingo with a chewed-up Rosetta Stone box, you would be less than pleased, despite my protestations that negative scientific results are useful and I had just proven that Spanish-illiterate dingoes cannot extinguish fires using mind power.

It was at this moment, when John had hit the bottom, that he discovered religion.

John began to attend The Church of the Impending Power Catastrophe. He sat in the pew and he heard the cautionary tales, and he was afraid. John learned about the new hyper-threaded processor from AMD that ran so hot that it burned a hole to the center of the earth, yelled "I've come to rejoin my people!", discovered that magma people are extremely bigoted against processor people, and then created the Processor Liberation Front to wage a decades-long, hilariously futile War to Burn the Intrinsically OK-With-Being-Burnt Magma People. John learned about the rumored Intel Septium chip, a chip whose prototype had been turned on exactly once, and which had leaked so much voltage that it had transformed into a young Linda Blair and demanded an exorcism before it embarked on a series of poor career moves that culminated in an inevitable spokesperson role for PETA. The future was bleak, and John knew that he had to fight it. So, John repented his addiction to scaling, and he rededicated his life to reducing the power consumption of CPUs. It was a hard path, and a lonely path, but John could find no other way. Formerly the life of the party, John now resembled the scraggly, one-eyed wizard in a fantasy novel who constantly warns the protagonist about the variety of things that can lead to monocular bescragglement. At team meetings, whenever someone proposed a new hardware feature, John would yell "THE MAGMA PEOPLE ARE WAITING FOR OUR MISTAKES." He would then throw a coffee cup at the speaker and say that adding new hardware features would require each processor to be connected to a dedicated coal plant in West Virginia. John's coworkers eventually understood his wisdom, and their need to wear coffee-resistant indoor ponchos lessened with time. Every evening, after John left work, he went to the bus stop and distributed power literature to strangers, telling them to abandon transistor scaling and save their souls. Standing next to John, another man wore a sandwich board that said that the Federal Reserve was using fluorinated water to hide the fact that we never landed on the moon. The sandwich board required no transistors at all. It made John smile.

When John comes home for the holidays, you're glad that he's back, but you miss the old twinkle in his eye. Your thoughts wander to your own glory days thirty years ago, when Aerosmith mistook young John for a large Xanax tablet and tried to trade him for a surface-to-air missile that could be used against anti-classic rock regimes. Oh, how you laughed! The subsequent visit by Child Protection Services was less amusing, but that was the way that hardware architects lived: working hard, partying hard, and occasionally waking up in Tijuana to discover that your left kidney is missing and your toddler has been shipped to a Columbian arms smuggler. It was crazy, but you wouldn't change a thing. Your generation had lived so many dreams, and slain so many foes.

Today, if a person uses a desktop or laptop, she is justifiably angry if she discovers that her machine is doing a non-trivial

amount of work. If her hard disk is active for more than a second per hour, or if her CPU utilization goes above 4%, she either has a computer virus, or she made the disastrous decision to run a Java program. Either way, it's not your fault: you brought the fire down from Olympus, and the mortals do with it what they will. But now, all the easy giants were dead, and John was left to fight the ghosts that Schrödinger had left behind. "John," you say as you pour some eggnog, "did I ever tell you how I implemented an out-of-order pipeline with David Hasselhoff and Hulk Hogan's moustache colorist?" You are suddenly aware that you left your poncho in the other room.