CFEngine 3.6

— Metadata —

Conference: LISA 2014

Author: Ted Zlatanov tzz@lifelogs.com

Audience: Anyone curious about CFEngine 3.6, with or without CFEngine 3 experience

— CFEngine Fundamentals —

Promise Theory

- read the book! *Promise Theory: Principles and Applications (Volume 1)* by Mark Burgess
- based on ideas from immunology, sociology, theoretical physics, computer science, and other fields
- autonomous agents
- promises are bidirectional
- idempotent operations
- convergent operations
- foundation for CFEngine the software, CFEngine the tool, CFEngine the knowledge framework…

Genealogy

In the 90's, Mark Burgess wrote cfengine 1 in C and people liked it. It worked. It was small and simple.

Later Mark Burgess wrote cfengine 2, and it was a lot like 1. Many saw this and wrote their own software, and some was like cfengine, and some wasn't. Thus the Puppet and Chef tribes arose, and they were good tools too, but had a Ruby requirement.

Mark Burgess wrote CFEngine 3, and it was not like 1 or 2, but was its own way, modular and powerful and still written in C. Meanwhile the Ansible and Salt tribes were formed and they were all alike but different enough to keep sales reps and developers employed.

CFEngine Enterprise and Community have been split since version 3. Enterprise version has Windows platform support, significantly better reporting, a GUI called the Mission Portal, and much more. Both are maintained by CFEngine AS. Community contributions have to go through Github pull requests. Enterprise is closed-source.

— CFEngine Primer —

Getting started with CFEngine is easy. You install according to The installation guide and off you go. It's especially easy to try it out with a recent version of Vagrant, which has the CFEngine provisioner built-in, and that's documented in https://docs.vagrantup.com/v2/provisioning/cfengine.html

Here's a minimal `Vagrantfile` you can use, although you may want to use a different OS:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "trusty"
  config.vm.box_url = "https://oss-binaries.phusionpassenger.com/vagrant/boxes/latest/ubuntu-14.04-amd64-vbox.box"
  config.vm.network "private_network", ip: "192.168.33.10"
  # config.vm.network "public_network"
  config.ssh.forward_agent = true
  # config.vm.synced_folder "../data", "/vagrant_data"

  config.vm.provision "cfengine" do |cf|
    cf.am_policy_hub = true
    # cf.run_file = "motd.cf"
  end
  #
  # You can also configure and bootstrap a client to an existing
  # policy server:
  #
  # config.vm.provision "cfengine" do |cf|
  #   cf.policy_server_address = "10.0.2.15"
  # end

end
```

You're all set, and you can verify that CFEngine is installed:

```
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Apr 22 19:47:09 2014 from 10.0.2.2
/usr/bin/xauth:  file /home/vagrant/.Xauthority does not exist
vagrant@ubuntu-14:~$ cf-agent -V
CFEngine Core 3.6.2
vagrant@ubuntu-14:~$ sudo ls /var/cfengine/inputs
cfe_internal  cf_promises_release_id  cf_promises_validated  controls  def.cf  failsafe.cf  inventory  lib  promises.cf  services  sketches  templates  update  update.cf

vagrant@ubuntu-14:~$ sudo cf-agent
```

There's no output, because there's nothing to do. Typically that's a good thing. Here's how you can get more information (note that `-KI` means "inform me, and re-run things even though they've been run recently"):

```
vagrant@ubuntu-14:~$ sudo cf-agent -KIC
```

(The `-C` is for color, and is really nice interactively.)

The following lines can be disabled, but they are harmless: they just trigger the package inventory to re-scan the package lists by trying to install a fake package.

```
2014-11-02T16:45:59+0000     info: /default/inventory_autorun/methods/'packages_refresh'/default/cfe_autorun_inventory_packages/packages: Installing cfe_internal_non_existing_package...
2014-11-02T16:45:59+0000     info: /default/inventory_autorun/methods/'packages_refresh'/default/cfe_autorun_inventory_packages/packages/'cfe_internal_non_existing_package'[0]: Q:env DEBIAN_FRON
TEND= ...:Reading package lists...
2014-11-02T16:45:59+0000     info: /default/inventory_autorun/methods/'packages_refresh'/default/cfe_autorun_inventory_packages/packages/'cfe_internal_non_existing_package'[0]: Q:env DEBIAN_FRON
TEND= ...:Building dependency tree...
2014-11-02T16:45:59+0000     info: /default/inventory_autorun/methods/'packages_refresh'/default/cfe_autorun_inventory_packages/packages/'cfe_internal_non_existing_package'[0]: Q:env DEBIAN_FRON
TEND= ...:Reading state information...
2014-11-02T16:45:59+0000     info: /default/inventory_autorun/methods/'packages_refresh'/default/cfe_autorun_inventory_packages/packages/'cfe_internal_non_existing_package'[0]: Q:env DEBIAN_FRON
TEND= ...:E: Unable to locate package cfe_internal_non_existing_package
R: cfe_autorun_inventory_packages: refresh interval is 10080
R: cfe_autorun_inventory_packages: we have the inventory files.
```

Here we see the system inventory from `dmidecode` and other info:

```
R: cfe_autorun_inventory_dmidecode: Obtained BIOS vendor = 'innotek GmbH'
R: cfe_autorun_inventory_dmidecode: Obtained BIOS version = 'VirtualBox'
R: cfe_autorun_inventory_dmidecode: Obtained System serial number = '0'
R: cfe_autorun_inventory_dmidecode: Obtained System manufacturer = 'innotek GmbH'
R: cfe_autorun_inventory_dmidecode: Obtained System version = '1.2'
R: cfe_autorun_inventory_dmidecode: Obtained CPU model = ''
R: inventory_linux: OS release ID = ubuntu, OS release version = 14_04
R: inventory_lsb: OS = Ubuntu, codename = trusty, release = 14.04, flavor = Ubuntu_14_04, description = Ubuntu 14.04 LTS
```

The `inventory_linux` line is getting the OS from file and command parsing, while the `inventory_lsb` line is using the LSB files and commands. They are duplicated here, but on a non-LSB Linux system you may have to use `inventory_linux`.

So far CFEngine is not doing anything for you, the user! All this, just to do nothing? Well, all this inventorying is what Chef and Puppet and Ansible and others do as well, albeit with specialized tools (Facter, Ohai, etc.). With CFEngine it's just done with the base policy, rather than a add-on tool.

If you want to customize the base policies, take a look in the file `/var/cfengine/masterfiles/def.cf` which has most of the defaults you'll need to set. But let's just do some example one-liner policies. These will be for version 3.6.2 or later, so make sure you didn't accidentally install something older.

We will not discuss the client-server setup, known as **bootstrapping**, but the CFEngine documentation has all that information. It's not hard, it's just not necessary to see how the software works.

— CFEngine 3.6 —

In 3.6, major additions to the base policies were done in **bundles** (for now, think of them as functions) to do common tasks. Before 3.6, these were added haphazardly, but in 3.6 the base set of bundles was greatly expanded. So we'll use them whenever possible. You can find the list at https://docs.cfengine.com/docs/3.6/reference-standard-library.html

Another improvement in 3.6 are the `--list-classes` and `--list-vars` options to `cf-promises`. Run with them to see the list of built-in variables and classes with the base policies. For example, `sys.libdir` will expand to `/var/cfengine/inputs/lib/3.6` in a 3.6 installation.

The starting point for our policies will be very simple:

```
body common control
{
    inputs => { "$(sys.libdir)/stdlib.cf" };
    bundlesequence => { main };
}

bundle agent main
{
  methods:
}
```

The `inputs` line tells CFEngine to include the standard library. So whatever we add from this point on, assume it will go in the `main` bundle after `methods:`, and run the example with `cf-agent -KIC -f example.cf`.

File operations

These are just examples, there are many more file-related bundles, from line-based editing to setting permissions. See https://docs.cfengine.com/docs/3.6/reference-standard-library-files.html for more.

Remove a file or directory to any depth

```
     "bye" usebundle => rm_rf("/var/tmp/oldstuff");
```

Remove a single file

```
     "bye" usebundle => file_tidy("/var/tmp/oldstuff.txt");
```

Empty a file, creating if needed

```
     "empty" usebundle => file_empty("/var/tmp/mystuff.txt");
```

Link a file (symbolic or hard link)

```
     "" usebundle => file_hardlink("/tmp/z.txt", "/tmp/z.link");
     "" usebundle => file_link("/tmp/z.txt", "/tmp/z.link");
```

Synchronize two directories, deleting unknowns

```
     "" usebundle => dir_sync("/tmp", "/var/tmp");
```

Make a file from a string

```
     "" usebundle => file_make("/tmp/z.txt", "Some text, libdir is $(sys.libdir),
and some more text here");
```

Note that you can use multiple lines and variable expansions.

Make a file from a Mustache template combined with JSON data

The Mustache template is in `x.mustache`. The JSON data is inline here, but can also be in a file (use `file_mustache` for that case). The output will be in `z.txt`.

```
     "m" usebundle => file_mustache_jsonstring("x.mustache", '{ "x": "y" }', "z.txt");
```

In CFEngine 3.6, Mustache templates are both simple and powerful. See the documentation for details on how to use them.

Packages

These bundles are essential and there was nothing like them in 3.5 or earlier. See https://docs.cfengine.com/docs/3.6/reference-standard-library-packages.html for more; you can also target a specific version or install from a file (e.g. on Solaris).

It's very likely that the underlying `packages` promises will be changed or deprecated in a future version of CFEngine, but these convenient bundles will remain.

Install a package

```
"pleasezip" usebundle => package_present("zip");
```

Install and keep a package up to date

```
"pleasezip" usebundle => package_latest("zip");
```

Remove a package

```
"nozip" usebundle => package_absent("zip2");
```

And many more bundles…

The full list of convenience bundles is very long, so just keep in mind that if you want to do something in CFEngine, your first step should be to look for a convenience bundle that does it.

Users promises

You can add, reconfigure, and remove users. Isn't that awesome? See https://docs.cfengine.com/docs/3.6/reference-promise-types-users.html for the details and usage examples.

Data containers

We mentioned earlier you can make a file from JSON data applied to a Mustache template, but how does it work?

The magic is in a new variable type called `data`. It's a container that happens to look and act like JSON internally, but can be treated like a CFEngine array variable at the policy level. Well, here is the `datatype.cf` example:

```
  vars:
      # load at most 1MB from datatype.cf.json
      "config" data => readjson("$(this.promise_filename).json", 1024k);
      # ... or inline ...
      # "config" data => parsejson('[ ... JSON data here ... ]');

      # get the keys of the config
      "keys" slist => getindices(config);

      # get the second-level keys
      "second_$(keys)" slist => getindices("config[$(keys)]");

  reports:
      # don't be scared, this is just variable substitution and implicit iteration!
      "$(this.bundle): from the loaded config, we got top-level key $(keys) and under it, second level key $(second_$(keys)) with value $(config[$(keys)][$(second_$(keys))])";
```

The JSON data:

```
[
    {
        "name": "firstkey",
        "x": 100,
        "y": 200
    },
    {
        "name": "secondkey",
        "x": 4500,
        "y": 9900
    },
]
```

The output:

```
R: main: from the loaded config, we got top-level key 0 and under it, second level key name with value firstkey
R: main: from the loaded config, we got top-level key 1 and under it, second level key name with value secondkey
R: main: from the loaded config, we got top-level key 0 and under it, second level key x with value 100
R: main: from the loaded config, we got top-level key 0 and under it, second level key y with value 200
R: main: from the loaded config, we got top-level key 1 and under it, second level key x with value 4500
R: main: from the loaded config, we got top-level key 1 and under it, second level key y with value 9900
```

If you're used to the CFEngine 3.5 and older array variables, you'll recognize the addressing format with square brackets. You should simply use the `getindices` and `getvalues` functions that you used for array variables without fear. The `mergedata` function is the only remaining piece of the puzzle: when passed two or more data containers, it will merge them from left (bottom) to right (top), overwriting any keys on the bottom. So `{ x: 100 }` merged on top of `{ x: 200 }` will give you `{ x: 100 }`. But merging JSON arrays or slists will append them, as you'd expect.

Finally, there's the `storejson` function to write a data container into a JSON data file.

There are also functions to read a data container from various data formats, e.g. simple comma-separated lists or key-value lists. Explore.

format() and eval() functions

The new `format` and `eval` functions let you print formatted data, including data containers, and evaluate mathematical expressions. Here's a fairly complete example in `eval.cf`.

```
  vars:
      "values[0]" string => "x"; # bad
      "values[1]" string => "+ 200"; # bad
      "values[2]" string => "200 + 100";
      "values[3]" string => "200 - 100";
      "values[4]" string => "- - -"; # bad
      "values[5]" string => "2 + 3 - 1";
      "values[6]" string => ""; # 0
      "values[7]" string => "3 / 0"; # inf but not an error
      "values[8]" string => "3^3";
      "values[9]" string => "-1^2.1"; # -nan but not an error
      "values[10]" string => "sin(20)";
      "values[11]" string => "cos(20)";
      "values[19]" string => "20 % 3"; # remainder
      "values[20]" string => "sqrt(0.2)";
      "values[21]" string => "ceil(3.5)";
      "values[22]" string => "floor(3.4)";
      "values[23]" string => "abs(-3.4)";
      "values[24]" string => "-3.4 == -3.4";
      "values[25]" string => "-3.400000 == -3.400001";
      "values[26]" string => "e";
      "values[27]" string => "pi";

      "indices" slist => getindices("values");

      "eval[$(indices)]" string => eval("$(values[$(indices)])", "math", "infix");

  reports:
      "math/infix eval('$(values[$(indices)])') = '$(eval[$(indices)])'";
```

Output from the good expressions (the bad ones produce no result):

```
R: math/infix eval('+ 200') = ''
R: math/infix eval('20 % 3') = '2.000000'
R: math/infix eval('pi') = '3.141593'
R: math/infix eval('200 - 100') = '100.000000'
R: math/infix eval('-1^2.1') = '-nan'
R: math/infix eval('-3.400000 == -3.400001') = '0.000000'
R: math/infix eval('- - -') = ''
R: math/infix eval('e') = '2.718282'
R: math/infix eval('sqrt(0.2)') = '0.447214'
R: math/infix eval('sin(20)') = '0.912945'
R: math/infix eval('') = '0.000000'
R: math/infix eval('ceil(3.5)') = '4.000000'
R: math/infix eval('abs(-3.4)') = '3.400000'
R: math/infix eval('2 + 3 - 1') = '4.000000'
R: math/infix eval('x') = ''
R: math/infix eval('floor(3.4)') = '3.000000'
R: math/infix eval('-3.4 == -3.4') = '1.000000'
R: math/infix eval('3^3') = '27.000000'
R: math/infix eval('200 + 100') = '300.000000'
R: math/infix eval('3 / 0') = 'inf'
R: math/infix eval('cos(20)') = '0.408082'
```

Other new functions

All these are documented in https://docs.cfengine.com/docs/3.6/guide-latest-release-whatsnew.html with links to their documentation and examples.

- `bundlesmatching`: list all the bundles matching a regular expression and optionally a set of tags. That lets users construct dynamic policies.
- `classesmatching` and `variablesmatching`: list all the classes or variables matching a regular expression and optionally a set of tags. Extremely useful for discovering what you actually know.
- `findfiles`: find files with standard glob patterns.
- and much more… statistics, data access, string operations…

Core improvements

- new TLS protocol
- LMDB as the data storage backend
- `shortcut` attribute to serve secret files only to specific clients
- policy tagging with `cf-promises -T` that will use the Git or Subversion revision, or generate a tree checksum

— Design Center —

Along with CFEngine 3.6, you will find the Design Center. This is an optional module repository where you can find **sketches** to implement common system policies. Think of them as extensible, reusable bundles with an API. They are used in CFEngine Enterprise and you can write your own.

In CFEngine Community, the Design Center requires command-line interaction and it's highly recommended that you also deploy your policies from Git to make it easy to roll back and review your changes.

— Questions? —