Get set for getting work done by CFEngine

Sources:

- Brian Bennets Primer: https://github.com/bahamat/cf-primer
- Nick Andersons Primer: https://github.com/nickanderson/CFEngine-zero-to-hero-primer
- Neil Watsons Autorun-Primer: https://digitalelf.net/2014/07/a-primer-on-cfengine-3-dot-6-autorun

I changed and mixed them a bit to become a more practical introduction and kept things out that are not important to get started.

Introduction

Why did I start using CFEngine? I liked the idea of giving away the control and all the detailled knowledge about the different Linux distributions and instead just declaring a state and let some software doing the rest.

CFEngine contains a powerful language for controlling all aspects of a system. CFEngine runs primarily on UNIX und UNIX like OS, but can run on Windows also. In this presentation you will just learn a subset of what CFEngine can do but this will show you how easy it is to get the most of the work done.

If you have any question later or want a training on CFEngine please feel free to contact me via info@linden-it-net.de

Let's start with the four components that make the piece of software:

- cf-agent
- cf-monitord
- cf-execd
- cf-serverd

cf-agent

cf-agent is the command you will use most often. It is used to apply policy to your system. If you are running any CFEngine command from the command line, it will be cf-agent.

cf-monitord

It monitors various statistics about the running system and makes this information available in the form of classes and variables. You will almost never use cf-monitord directly. The data provided by cf-monitord is available to cf-agent.

cf-execd

It's a periodic task scheduler and it takes care of the policies getting run every five minutes by default.

cf-serverd

It runs on the CFEngine server as well as on all clients.

- On a server it's responsible for serving files to clients.
- On a client it accepts cf-runagent requests

cf-runagent allows you to request ad-hoc policy runs. I didn't use it yet.

BIG Difference

.. in thinking and doing

Imperative vs. Declarative

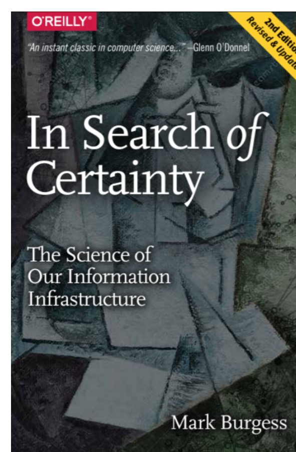
By using CFEngine we don't tell the machine (or interpreter) what it has or what we want it to do. We use the CFEngine coding language to describe a state we want a system to have. We just describe the state, we do not code how to get there. For example a state can be:

- The file /etc/motd contains the single line: "This server is managed by CFEngine".
- The apache webserver is installed
- The apache webserver is running (and is getting restarted if it crashed for some reason)
- The httpd.conf is this or that or contains this or another line or setting
- The owner and group of the webserver directory is webowner:webgroup

This method offers the possibility to detect any deviation from this described and desired state + knowing what needs to be done to transform the actual state to the desired state. Further the process can start in any state, not just in a known one. CFEngine uses convergence to arrive at the decribed state. When a system has reached the desired state it is said to have reached convergence.

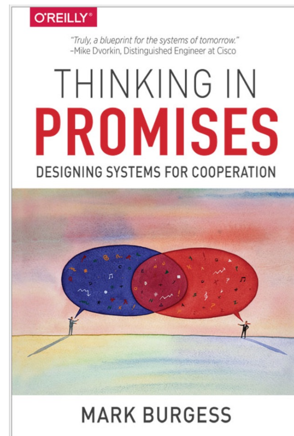
Philosophy / Promise Theorie

Promise theory is the fundamental underlying philosophy that drives CFEngine. It is a model of voluntary cooperation between individual, autonomous actors or agents who publish their intentions to one another in the form of promises. There is an incredible deep and scientific background of CFEngine. After six months I finished one-third of:



.. and it's absolutely thrilling!

There're other books of the CFEngine founder Mark Burgess available as well like:



What makes promises?

- A file (e.g., /etc/apache2/httpd.conf) can make promises about its own contents, attributes, etc. But it does not make any promises about a process.
- A process (e.g., httpd) can make a promise that it will be running. But it does not make any promises about its configuration.
- The configuration file and the process are autonomous. Each makes promises about itself which cooperates toward an end.

Anatomy of a Promise

```
type:
  context::
    "promiser" -> "promisee"
      attribute1 => "value1",
      attribute2 => "value2";
```

- type is the kind of promise being made; e.g. files, commands, packages, processes, services, storage, users, ..
- context is optional; in the code there're classes/class guards at this point acting like if/thens. Promises with this context/class will only apply if the given context is true resp. the given class is set -> see example below.
- promiser is what is making the promise; e.g. a file, a command, a package, a process, a service, a storage, a user, ..
- promisee is an optional recipient or beneficiary of the promise; they are useful to generate reports about how often a promise was run and what the outcomes have been.

For this introduction I leave the promisee out; I use CFEngine since 18 months and never used any promisee yet as reports about separate promises were not requested by my employer.

Example of a Promise

```
files:
  sles.ipv4_10_4_1::
    "/etc/conf.file"
      create  => "true",
      perms => mog("644","root","other");
```

The promise type is "files" and refers to the file "/etc/conf.file". The promise is read by the cf-agent on the node and will only

apply if it's a SLES box (Suse Linux Enterprise Server) AND (.) has an IPv4 address in the 10.4.1 subnet. If this is the case the file will be created (for the case it doesn't exist yet) and the permissions will be set to 644, the file owner will be "root" and the file owning group will be "other". If the node is a redhat box, even if it's the 10.4.1.10, the promise will not apply; same if it's a SLES box but in another subnet, the promise won't apply. If it's neither SLES nor has an IP in the given subnet it won't apply either.

Promise Attributes

Each promise can have one or more attributes that describe the parameters of the promise. The available attributes will vary depending on the promise type. The value can be either a text string (which must be quoted) or another object (which must not be quoted). All attributes together are called the body of the promise. Attributes are separated by commas. Each promise ends with semi-colon.

Example of a Promise with another Object as Attribute

```
files:
  sles.ipv4_10_4_1::
    "/etc/conf.file"
      create  => "true",
      perms => mog("644","root","other")
      edit_line => check4value;
```

The edit_line attribute (and the perms attribute as well) is a bundle (more about bundles in a minute) that is shipped with any CFEngine package (Community and Enterprise) in its library; just use it/them! "check4value" is a name I chosed for my own object to call/use the edit_line bundle:

```
bundle edit_line check4value
{
    classes:
        "value_is_set" expression => regline("MyConfigValue: enabled", "$(edit.filename)");
}
```

With this object as attribute it's checked if "MyConfigValue: enabled" is found in /etc/conf.file. As my object bundle is called out of a files-promise the file which is promised about is being placed in the special variable $(edit.filename), because this is the file getting edited. If "MyConfigValue: enabled" is found in the file the class resp. the context "value_is_set" is being set; given that the node is a linux box.

The promise type in this edit_line bundle is "classes" and the class "value_is_set" promises to be set if the given expression evaluates to true. If not the class won't be set.

Given it's set, we can use this class/context for another decision. Again: classes/context are the if/thens of CFEngine.

This is a bundle with just one (classes-)promise, but mostly...

Bundles

.. are collections of promises. It is a logical grouping of any number of promises, usually for a common purpose. E.g. a bundle to configure everything necessary for Apache to function properly. Such a bundle might:

- install the apache2 package (packages promise)
- edit the configuration file (files promise)
- copy the web server content (files promise)
- configure filesystem permissions (files promise)

- ensure the httpd process is running (processes promise)
- restart the httpd process when necessary (processes promise)

Anatomy of a Bundle

```
bundle type name
{
  type:
    context::
      "promiser" -> "promisee"
      attribute1 => "value1",
      attribute2 => "value2";

  type:
    context::
      "promiser" -> "promisee"
      attribute1 => "value1",
      attribute2 => "value2";
}
```

Bundles apply to the binary that executes them. E.g., agent-bundles apply to cf-agent while server-bundles apply to cf-serverd.

Bundles of type common apply to any CFEngine binary. For now you will only create agent or common bundles.

Bodies

I stated before that the attributes of a promise, collectively, are called the body. Depending on the specific attribute the value of an attribute can be an external body. A body is a collection of attributes. These are attributes that supplement the promise.

Anatomy of a Body

```
body type name
{
    attribute1 => "value";
    attribute2 => "value";
}
```

The difference between a bundle and a body is that a bundle contains promises while a body contains only attributes.

Take a moment to let this sink in.

- A bundle is a collection of promises.
- A body is a collection of attributes that are applied to a promise.

The distinction is subtle, especially at first and many people are tripped up by this.

In a body each attribute ends with a semi-colon.

Abstraction and Re-usability

Bundles and bodies can be parameterized for abstraction and re-usability. You can define bundles and bodies and call them passing in parameters which will implicitly become variables.

An example will help:

```
bundle agent add_users
{
    methods:
      "create_users" usebundle => create_users("add_users.users");
```

```
    vars:
      # 1rst User
      "users[new_user1][uid]"               string => "6666";
      "users[new_user1][description]"       string => "NewUser1, Department, Company";
      # 2nd User
      "users[new_user2][uid]"               string => "6667";
      "users[new_user2][description]"       string => "NewUser2, Department, Company";
      # 3rd User
      "users[new_user3][uid]"               string => "6668";
      "users[new_user3][description]"       string => "NewUser3, Department, Company";

}
```

In this bundle we define the users in an array and call another bundle via a method + passing the users-array in the call of the bundle which will create the defined users. CFEngine iterates over all elements in the array.

Here's the called bundle:

```
bundle agent create_users(info)
{
    vars:
      "user"         slist  => getindices("$(info)");

    classes:
      "add_${user}"  not => userexists( ${user} );

    users:
      "add_${user}"::
        "$(user)"
          uid => " $($(info)[$(user)][uid])",
          policy => "present",
          description => "$($(info)[$(user)][description])",
          home_dir => "/home/$(user)",
          home_bundle => setup_home_dir("$(user)"),
          group_primary => "users",
          password => ThePassword,
          shell => "/bin/bash",
            classes => if_repaired("user_added");

    reports:
      user_added::
        "User $(user) was added.";
}

bundle agent setup_home_dir(user)
{
    files:
      "/home/$(user)/." create => "true";
}

body password ThePassword
{
    format => "hash";
    data => "jNKlcaMQIuqBY"; # "CFEngine" # generated via 'openssl passwd'
}
```

What happens:

- vars: In the var "user" the indices of the array are stored: new_user1, new_user2 and new_user3.
- classes: a class "add_${user}" is added if the user doesn't exist on the system.
- users: Just if the class "add_${user}" is set (add_new_user1, add_new_user2 or add_new_user3) the user promise will be run. If the user already exists (the promise was run before) nothing will happen.
- reports: Just if the class "user_added" is set (it will just be set if a user was added, look last line user-promise) the report will be printed. Reports are printed into /var/log/messages by default; they can be redirected into a file as well.
- If the users-promise is run the home_bundle attribute of the users-promise-type will call the setup_home_dir bundle with the users name as parameter. The setup_home_dir-bundle will create the user's home directory (The /. creates the directory).
- Similar to the latter: If the users-promise is run the password attribute of the users-promise-type will call the the password-body. Remember: A body just contains attributes that are applied to a promise. The password-body is shipped with the library and it knows how to deal

with it. All I've to do is to paste my hash in as data value
which I generated using "openssl passwd CFEngine".
This will result in the initial password set to "CFEngine"
for the new user.

Running Bundles

To run a single bundle for testing purposes do:

```
cf-agent -KIf yourPromise.cf -b yourBundle
```

```
* -K remove any locks
* -I inform mode, show me what happened; -v is too much for short information
* -f for file and
* -b for bundle
```

Using the -b option you can run just one bundle for the case you have several in your promise file.
We can define a bundlesequence using the -b option:

```
cf-agent -KIf yourPromise.cf -b bundle1,bundle2,bundle3
```

To run your whole policy:

```
cf-agent
```

That's it. To get an ouput:

```
cf-agent -v
```

This will run all your bundles.

Running bundles this way is not how it works in production. How do we activate our bundles for the cf-agent? We have a couple of choices:

1. /var/cfengine/masterfiles/promises.cf
2. via methods in main.cf
3. via meta-tags (autorun feature)
4. augment-/def.json-file

Which one is best to use depends on the environment and what you want to achieve.

Using 1 or 2 you need to re-configure your promises and files into the config-files after an upgrade.
Using the autorun-feature or the augment-/json-file (def.json) makes your upgrades easier, because you don't need to touch the masterfiles (MPF - Masterfiles Policy Framework), which may change/get overwritten by an upgrade.

The autorun works using meta-tags. If you use in a promise:

```
bundle agent MyBundle
{
  meta:
      "tags" slist =>  { "autorun" };
...
..
}
```

.. this bundle will be run automatically by the cf-agent.

In the example I'm demonstrating we include our promise file resp. bundle "security_bundle.cf" in the def.json which looks this way:

```
[root@hub masterfiles]# cat def.json
{
"inputs": [ "$(sys.workdir)/inputs/getset/security_bundle.cf" ],
}
```

If we put all our policy files in the def.json there's no need to touch the shipped masterfiles after an upgrade. Current versions of CFEngine will check for a def.json (in /var/cfengine/inputs/promises.cf).

This way it's ensured that the promise file will be found and parsed BUT NOT actuated. To actuate it I use the following /var/cfengine/inputs/services/main.cf:

```
bundle agent main
# User Defined Service Catalogue
{
  vars:
    "bundles" slist => bundlesmatching(".*","security");

  reports:
    "I found security bundles: $(bundles)";

methods:
# Activate your custom policies here
  "" usebundle => $(bundles);
}
```

This main.cf finds the security_bundle.cf tagged by "security":

```
bundle agent security_bundle
{
  meta:
    "tags" slist => { "security" };

  reports:
    "Hello from $(this.bundle)";
}
```

.. and runs it due to the method above. This way bundles can be tagged and found by tag and fully dynamic bundle sequences can be constructed.

The def.json file can contain your (global) defined classes, variables (e.g. your ACLs) and input files, that are your promises you want to get run by CFEngine's cf-agent.

Updating policies on the nodes (manually)

All couple of minutes the nodes check for promise updates on the hub automatically. If you have new promises on the hub and want to get them pulled by the nodes immediately do it by:

```
cf-agent -KIf /var/cfengine/inputs/update.cf -D validated_updates_ready
```

Debugging

There will be errors.. and w/o knowing where to look you are lost. CFEngine has a -v (verbose) log and it's really a verbose one. You can always initiate an agent run by "cf-agent -v". As this is verbose indeed it makes sense to put that output into a file by "cf-agent -v > /tmp/cf-agent.out". In this cf-agent.out you can search for your bundle, for your classes and reports that just get printed if a condition/class was set or not set.

If you wonder why this or that promise isn't run and you do miss a result on your target node/s.. .. it's helpful to use reports to see if your conditions resp. classes are set.

Important things to consider

Regarding the deployment of new nodes in "your" environment. What is the best way to organize your nodes, means how to identify them by classes? By ip, by network, by name, by architecture, by OS, by any software installed? How will CFEngine know: this new server is for this purpose (Webserver, DB-Server, File-Server, for whatever department, ..) The answer for this depends on the environment.

There are tricky ways to actuate bundles depending on the existance of (hard or soft) classes.

What more is possible to do by using CFEngine

CFEngine has more power:

- Virtualization / Guest Environments: Promise the existence of virtual machines.
- Storage: Promise local or remote (NFS) filesystems.
- Database: Promise the schema of your database, CFEngine does the SQL for your you.
- Interfaces: Promise your network settings
- Monitoring: Using data from cf-monitor.
- Reporting: Report whatever, based on classes set or unset: - A process/service running on which machines in which network/s (set a class if a process crashes and report this way how often a process crashes on what systems) - What application is installed in what version on what nodes. - .. uncountable possibilities, pretty easy to set up

Reports can be written to files. CFEngine-Enterprise comes with a GUI where reports can be created and shown graphically.

Support

For any questions contact me on: lindomatic@gmail.com or reach the community at https://groups.google.com/forum/#!forum/help-cfengine