cfengine tips

sipXecs 4.6 now bundles the open source tool called "cfengine" to apply configuration to a system and manage services. This is a role that a C++ service called "sipxsupervisor" previously filled. sipxsupervisor will continue to be installed and used for until all management is converted to cfengine scripts.

cfengine has many advantages over sipxsupervisor including:

- scripts you can develop without compiling
- rich libraries of scripts to perform common operations
- documentation and written books
- support for idempotent operations.
- scripts that execute then exit so cannot leave memory leaks
- reporting
- key based security built in

If your company's IT infrastructure already uses a configuration tool like puppet, chef or even cfengine, it should not conflict with the cfengine used by sipXecs. This includes the situation when both services are changing the same file! There would only be conflict if both systems are trying to change the same settings to different values and in that case you can modify the cfengine scripts that are installed but sipXecs if conflict cannot be resolved.

Here are some helpful tips with working with cfengine.

Online documentation

1. Reference Manual - Best doc for explaining what functions are built into cfengine
2. Tutorial - Goes into theory quickly so I don't think it makes the greatest tutorial but you may find some useful explanations however. Much of it regarding how servers work together has already be integrated into sipXecs, or simply does not apply to how sipXecs uses cfengine.
3. Standard cfengine Library - There's a library of utilities we including in sipXecs that are available to your scripts.

How to develop cfengine scripts independently

Like a shell script, many times it makes sense to develop your cfengine script outside of sipXecs. Here is the simplest script you could write, and how you could execute it.

/var/cfengine/inputs/mytest.cf

```
# this is a comment
bundle agent mytest {
  vars:
    any::
      "foo" string => "bar";

  commands:
    any::
      "/bin/touch"
        args => "/tmp/bar";

  classes:
    any::
      "myflag" expression => fileexists("/tmp/foobar");

  files:
    any::
      "/tmp/foo"
        create => "true";

  reports:
    linux::
      "foo variable is $(foo)";
    myflag::
      "/tmp/foobar does exist";
    !myflag::
      "/tmp/foobar does *not* exist";
}

body common control {
  bundlesequence => {
    "mytest"
  };
}
```

And how to run script

```
cd /var/cfengine/inputs
cf-agent -IKvf mytest.cf
```

"classes" are just booleans

If you're coming from a programming background, classes usually mean OOP. In cfengine, think of them as booleans or flags you define.

Inside an agent "bundle" you can use any of the following "promise types"

1. vars
2. classes
3. outputs
4. interfaces
5. files
6. packages
7. environments
8. methods
9. processes
10. services
11. commands
12. storage
13. databases
14. reports

They are "executed" in this order. This is very important. Normally the order is exactly what you want, but sometimes it is not. For example you may want a promise in the "commands" section to execute **before** a promise in "files" section. When this is true, the most straightforward way to to break your bundle into two separate bundles and then change the order in the "inputs" order. Unfortunately your variables are now spread across the two bundles.

Here's an example

Before breakup

```
bundle agent foo {
  files:
    "/some/file"
      classes => if_repaired("do_this");

  #  BROKEN!!  because files is executed *after* classes
  classes:
    do_this::
       "bar" expression => fileexists("/some/other/file");
}
```

After breakup

```
bundle agent foo {
  files:
    "/some/file"
      classes => if_repaired("do_this");

  methods:
    do_this::
      "any" usebundle => "bar";
}

bundle agent bar {
  classes:
    # this works only because "if_repaired" defines "do_this" in global context
    # otherwise you'd have to use "$(foo.do_this)"
    do_this::
       "bar" expression => fileexists("/some/other/file");
}
```

Create an empty directory

No obvious, but here's how

```
bundle agent generate_certs {
  files:
    any::
      "$(sipx.SIPX_VARDIR)/temp/cert-temp/."
         create => "true";
}
```

Control order of promises.

All the plugins are executed in alphabetical order in /use/share/sipxecs/cfinputs/plugin.d but sometimes you want to ensure a promise is executed before your promise and you'd rather not have to change the name of your modules. All you need to do is use any bundles that you require to be run before your module. Those modules will only be run once, just before your module.

Example:

aaaa.cf

```
bundle agent aaaa {
  methods:
    "any" usebundle => zzzz;
  reports:
    any::
      "This will get executed after zzzz but only once";
}
```

zzzz.cf

```
bundle agent zzzz {
  reports:
    any::
      "This will get executed before aaaa_otherstuff_after_zzzz but only once";
}
```