Promising and Editing File Content

A CFEngine Special Topics Handbook

CFEngine AS

The ability to edit files convergently is a popular and widely used aspect of CFEngine. This document proposes some best practices for managing file content.

The examples contained here assume the inclusion of the standard COPBL library as an input.

Table of Contents

From boiler-plates to convergent file editing

Many configuration management systems allow you to determine configuration file content to some extent, usually by over-writing files with boiler-plate (template) files. This approach works for some cases, but it is a blunt and inflexible instrument, which forces you to take over the ownership of the file 'all or nothing' and determine its entire content yourself. This is more than is necessary or desirable in general.

Other approaches to file editing us search and replace, e.g. with the long-standing Unix tools '`awk`' and '`sed`'. Adding a user to a structured file such as the password file, only if the user is not already defined, is a more complex operation.

Cfengine allows you to model both whole files and parts of files, in any format, and promise that these fragments will satisfy certain promises about their state. This is potentially different from more common templating approaches to file management in which pre-adjusted copies of files are generated for all recipients at a single location and then distributed.

The most important thing about making changes to files is that the result end up being predictable. There are three ways to approach this problem. You should choose the simplest approach that solves your problem and try not to be prejudiced by what you have done before.

Why is file editing difficult?

File content is not made up of simple data objects like permission flags or process tables: files contain compound, ordered structures (known as grammars) and they cannot always be determined from a single source of information. To determine the outcome of a file we have to adopt either a fully deterministic approach, or live with a partial approximation.

Some approaches to file editing try to 'know' the intended format of a file, by hardcoding it. If the file then fails to follow this format, the algorithms might break. CFEngine gives you generic tools to be able to handle files in any line-based format, without the need to hard-code specialist knowledge about file formats.

Remember that all changes are adapted to your local context and implemented at the final destination by `cf-agent`.

What does file editing involve?

There are several ways to approach desired state management of file contents:

1. Copy a finished file template to the desired location, completely overwriting existing content.
2. Copy and adapt an almost finished template, filling in variables or macros to yield a desired content.
3. Make corrections to whatever the existing state of the file might be.

There are advantages and disadvantages with each of these approaches and the best approach depends on the type of situation you need to describe.

For the approach	**Against the approach**
1. Deterministic.	Hard to specialize the result and the source must still be maintained by hand.
2. Deterministic.	Limited specialization and must come from a single source, again maintained by hand.
3. Non-deterministic/partial model.	Full power to customize file even with multiple managers.

Approaches 1 and 2 are best for situations where very few variations of a file are needed in different circumstances. Approach 3 is best when you need to customize a file significantly, especially when you don't know the full details of the file you are starting from. Approach 3 is generally required when adapting configuration files provided by a third party, since the basic content is determined by them.

Three approaches to managing files

Copying a finished file template into place

Use this approach if a simple substution of data will solve the problem in all contexts.

1. Maintain the content of the file in a version controlled repository.
2. Check out the file into a staging area.
3. Copy the file into place.

```
bundle agent something
{
files:

  "/important/file"

  copy_from => secure_cp("/repository/important_file_template","svn-host");
}
```

Contextual adaptation of a file template

There are several approaches here:

1. Encode the boiler-plate template directly in the CFEngine configuration, and have full use of the power of the CFEngine language to adapt it.
2. Keep a separate boiler-plate file and edit/adapt it.
3. Copy a template from a repository then edit/adapt it.
4. Copy a generic template with embedded variables that can be expanded like macro-substitution.

Choose the approach that you consider to be simplest and most reliable for the purpose you need. Don't use templating, for instance, simply because it is what you are used to, or you might waste a lot of time and effort maintaining data that you don't need to.

To expand a template file on a local disk:

```
bundle agent templating
{
files:

  "/home/mark/tmp/file_based_on_template"

       create => "true",
    edit_line => expand_template("/tmp/source_template");
}
```

As of CFEngine version 3.3.0 you can also use a new templating file format and write:

```
bundle agent templating
{
files:

  "/home/mark/tmp/file_based_on_template"

       create => "true",
    edit_template => "/tmp/source_template";
}
```

For example, the source template file might look like this, with embedded CFEngine variables:

```
     mail_relay = $(sys.fqhost)
     important_user = $(mybundle.variable)
     #...
```

These variables will be filled in by CFEngine assuming they are defined within your CFEngine configuration.

If you use the new `edit_template` promise, you can embed directives to CFEngine context-classes and mark out regions of a file to be treated as an iteratable block.

```
#This is a template file /templates/input.tmpl

These lines apply to anyone

[%CFEngine solaris.Monday:: %]
Everything after here applies only to solaris on Mondays
until overridden...

[%CFEngine linux:: %]
Everything after here now applies now to linux only.

[%CFEngine BEGIN %]
This is a block of text
That contains list variables: $(some.list)
With text before and after.
[%CFEngine END %]

nameserver $(some.list)
```

For example: if we use this template in a promise:

```
bundle agent test
{
vars:
 "var" slist => { "1", "2", "3"};

files:
  "/tmp/expander"
         create => "true",
  edit_template => "/templates/input.tmpl";
}
```

The result would look like this, on a linux host:

```
#This is a template file /templates/input.tmpl

These lines apply to anyone
Everything after here now applies now to linux only.
This is a block of text
That contains list variables: 1
With text before and after.
This is a block of text
That contains list variables: 2
With text before and after.
```

```
This is a block of text
That contains list variables: 3
With text before and after.
nameserver 1
nameserver 2
nameserver 3
```

Example file template

```
[%CFEngine any:: %]
<VirtualHost $(sys.ipv4[eth0]):80>
        ServerAdmin             $(stage_file.params[apache_mail_address][1])
        DocumentRoot            /var/www/htdocs
        ServerName              $(stage_file.params[apache_server_name][1])
        AddHandler              cgi-script cgi
        ErrorLog                /var/log/httpd/error.log
        AddType                 application/x-x509-ca-cert .crt
        AddType                 application/x-pkcs7-crl    .crl
        SSLEngine               off
        CustomLog               /var/log/httpd/access.log
</VirtualHost>

[%CFEngine webservers_prod:: %]
[%CFEngine BEGIN %]
<VirtualHost $(sys.ipv4[$(bundle.interfaces)]):443>
        ServerAdmin             $(stage_file.params[apache_mail_address][1])
        DocumentRoot            /var/www/htdocs
        ServerName              $(stage_file.params[apache_server_name][1])
        AddHandler              cgi-script cgi
        ErrorLog                /var/log/httpd/error.log
        AddType                 application/x-x509-ca-cert .crt
        AddType                 application/x-pkcs7-crl    .crl
        SSLEngine               on
        SSLCertificateFile      $(stage_file.params[apache_ssl_crt][1])
        SSLCertificateKeyFile   $(stage_file.params[apache_ssl_key][1])
        CustomLog               /var/log/httpd/access.log
</VirtualHost>
[%CFEngine END %]
```

Combining copy with template expansion

What about getting your template to the end-host? To convergently copy a file from a source and then edit it, use the following construction with a staging file.

```
bundle agent master
{
files:
  "$(final_destination)"
         create => "true",
     edit_line => fix_file("$(staging_file)"),
 edit_defaults => empty,
         perms => mo("644","root"),
        action => if_elapsed("60");
}

bundle edit_line fix_file(f)
{
insert_lines:
  "$(f)"
     insert_type => "file";
     # expand_scalars => "true" ;

replace_patterns:
    "searchstring"
          replace_with => value("replacestring");
}
```

Making delta changes to someone else's file

Edit a file with multiple promises about its state, when you do not want to determine the entire content of the file, or if it is unsafe to make unilateral changes, e.g. because its contents are also being managed from another source like a software package manager.

For modifying a file, you have access to the full power of text editing promises. This is a powerful framework.

```
# Resolve conf edit

body common control
{
bundlesequence => { "g", resolver(@(g.searchlist),@(g.nameservers)) };
inputs => { "cfengine_stdlib.cf" };
}

bundle common g # global
{
vars:
 "searchlist"  slist => { "example.com", "cfengine.com" };
 "nameservers" slist => { "10.1.1.10", "10.3.2.16", "8.8.8.8" };

classes:
  "am_name_server"
     expression => reglist("@(nameservers)","$(sys.ipv4[eth1])");
}

bundle agent resolver(s,n)
{
files:
  "$(sys.resolv)"  # test on "/tmp/resolv.conf" #
      create        => "true",
      edit_line     => doresolv("@(this.s)","@(this.n)"),
      edit_defaults => empty;
}

# For your private library ......................

bundle edit_line doresolv(s,n)
{
insert_lines:
  "search $(s)";
  "nameserver $(n)";
delete_lines:
 # To clean out junk
  "nameserver .*| search .*" not_matching => "true";
}
```

Constructing files from promises

Making finished templates for files and filling in the blanks using variables is a flexble approach in many cases, but it is not flexible enough for all cases. A very flexible approach, but one that requires more thought, is to build a final result (desired end-state) from a set of promises about what the file should contain. This might or might not include templates in the sense of complete files that are read in.

If you are using CFEngine 3.3 or later, you have the option of using `edit_template` and its embedded language constructs to keep decisions and loops inside templates. Let's set aside that for a while and look at the alternatives, placing the data entirely within bundles of 'edit'-promises.

There is language support for this kind of editing in the standard library, and you can store data and template components within a CFEngine configuration itself, or as a separate file. For example:

```
#

body common control
{
bundlesequence => { "main" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

#

bundle common data
{
vars:
  "person" string => "Mary";
  "animal" string => "a little lamb";
}

#

bundle agent main
{
files:
   "/tmp/my_result"
        create => "true",
     edit_line => expand_template("/tmp/my_template_source"),
 edit_defaults => empty;
}
```

Suppose the file '`my_template_source`' contains the following text:

```
This is a file template containing variables to expand

e.g $(data.person) had $(data.animal)
```

Then we would have the file content:

```
host$ more /tmp/my_result
```

```
        This is a file template containing variables to expand

        e.g Mary had a little lamb
```

Adding a line here and there

A simple file like this could also be defined in-line, without a separate template file:

```
#

body common control
{
bundlesequence => { "main" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

#

bundle common data
{
vars:
  "person" string => "Mary";
  "animal" string => "a little lamb";
}

#

bundle agent main
{
vars:
  "content" string =>
    "This is a file template containing variables to expand
e.g $(data.person) had $(data.animal)";

files:

   "/tmp/my_result"
        create => "true",
     edit_line => append_if_no_line("$(content)"),
 edit_defaults => empty;
}
```

Lists inline

Here is a more complicated example, that includes list expansion. List expansion (iteration) adds some trickiness because it is an ordered process, which needs to be anchored somehow.

```
#
```

```
body common control
{
bundlesequence => { "main" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

#

bundle common data
{
vars:

  "person" string => "Mary";
  "animal" string => "a little lamb";

  "mylist" slist => { "one", "two", "three" };
  "clocks" slist => { "five", "six", "seven" };

  # or read the list from a file with readstringlist()

}

#

bundle agent main
{
files:


   "/tmp/my_result"

        create => "true",
     edit_line => my_expand_template,
 edit_defaults => empty;
}

#

bundle edit_line my_expand_template
{
vars:

 # import the lists, due to current limitation

 "mylist" slist => { @(data.mylist) };
 "clocks" string => join(", ","data.clocks");
 "other"  string => "eight";
insert_lines:

   "
   This is a file template containing variables to expand

   e.g $(data.person) had $(data.animal)

   and it said:
   ";

   "
   $(mylist) o'clock ";
   "
   ROCK!
   $(clocks) o'clock, $(other) o'clock
   ";

   "   ROCK!
   The end.
   "

   insert_type => "preserve_block"; # So we keep duplicate line
}
```

This results in a file output containing:

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ./test.cf -K
host$ more /tmp/my_result

   This is a file template containing variables to expand

   e.g Mary had a little lamb

   and it said:

   one o'clock
   two o'clock
   three o'clock
   ROCK!
   five, six, seven o'clock, eight o'clock
   ROCK!
   The end.
```

Splitting this example into several promises seems unnecessary and inconvenient, so we could use a special function `join()` to make pre-expand the scalar list and insert it as a single object:

```
#

body common control
```

```
{
bundlesequence => { "main" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

#

bundle common data
{
vars:

  "person" string => "Mary";
  "animal" string => "a little lamb";

  "mylist" slist => { "one", "two", "three", "" };
  "clocks" slist => { "five", "six", "seven" };

  # or read the list from a file with readstringlist()

}

#

bundle agent main
{
files:


   "/tmp/my_result"

        create => "true",
     edit_line => my_expand_template,
 edit_defaults => empty;
}

#

bundle edit_line my_expand_template
{
vars:

 # import the lists, due to current limitation

 "mylist" string => join(" o'clock$(const.n)  ","data.mylist");
 "clocks" string => join(", ","data.clocks");
 "other" string => "eight";
insert_lines:

   "
   This is a file template containing variables to expand

   e.g $(data.person) had $(data.animal)

   and it said:

   $(mylist)
   ROCK!
   $(clocks) o'clock, $(other) o'clock
   ROCK!
   The end.
   "

   insert_type => "preserve_block"; # So we keep duplicate line
}
```

Finally, since this is now entirely contained within a single set of quotes (i.e. there is a single promiser), we could replace the in-line template with one read from a file:

```
#

body common control
{
bundlesequence => { "main" };
inputs => { "LapTop/cfengine/copbl/cfengine_stdlib.cf" };
}

#

bundle common data
{
vars:

  "person" string => "Mary";
  "animal" string => "a little lamb";

  "mylist" slist => { "one", "two", "three", "" };
  "clocks" slist => { "five", "six", "seven" };

  # or read the list from a file with readstringlist()

}

#
```

```
bundle agent main
{
files:


   "/tmp/my_result"

        create => "true",
     edit_line => my_expand_template,
 edit_defaults => empty;
}

#

bundle edit_line my_expand_template
{
vars:

 # import the lists, due to current limitation

 "mylist" string => join(" o'clock$(const.n)  ","data.mylist");
 "clocks" string => join(", ","data.clocks");
 "other" string => "eight";

insert_lines:

   "/tmp/my_template_source"
     expand_scalars => "true",
     insert_type => "file";
}
```

Editing bundles

Unlike other aspects of configuration, promising the content of a single file object involves possibly many promises about the atoms within the file. Thus we need to be able to state bundles of promises for what happens inside a file and tie it (like a body-template) to the `files` promise. This is done using an `edit_line =>` or `edit_xml =>` constraint[1], for instance:

```
files:

  "/etc/passwd"

     create => "true",
```

[1] At the time of writing only `edit_line` is implemented.

```
     # other constraints on file container ...

     edit_line => mybundle("one","two","three");
```

Editing bundles are defined like other bundles for the agent, except that they have a type given by the left hand side of the constraint (just like body templates):

```
bundle edit_line mybundle(arg1,arg2,arg3)
{
insert_lines:

   "newuser:x:1111:110:A new user:/home/newuser:/bin/bash";
   "$(arg1):x:$(arg2):110:$(arg3):/home/$(arg1):/bin/bash";
}
```

Standard library methods for simple editing

You may choose to write your own editing bundles for specific purposes; you can also use ready-made templates from the standard library for a lot of purposes. If you follow the guidelines for choosing an approach to editing below, you will be able to re-use standard methods in perhaps most cases. Using standard library code keeps your own intentions clear and easily communicable. For example, to insert hello into a file at the end once only:

```
files:

  "/tmp/test_insert"

        create => "true",
     edit_line => append_if_no_line("hello"),
edit_defaults => empty;
```

Or to set the shell for a user

```
files:

  "/etc/passwd"
        create    => "true",
        edit_line => set_user_field("mark","7","/my/favourite/shell");
```

Some other examples of the standard editing methods are:

```
 append_groups_starting(v)
 append_if_no_line(str)
 append_if_no_lines(list)
 append_user_field(group,field,allusers)
 append_users_starting(v)
 comment_lines_containing(regex,comment)
 edit_line comment_lines_matching(regex,comment)
 delete_lines_matching(regex)
 expand_template(templatefile)
 insert_lines(lines)
```

```
resolvconf(search,list)
set_user_field(user,field,val)
set_variable_values(v)
set_variable_values2(v)
uncomment_lines_containing(regex,comment)
uncomment_lines_matching(regex,comment)
warn_lines_matching(regex)
```

You find these in the documentation for the COPBL.

Expressing `expand_template` as promises

As on CFEngine 3.3.0, CFEngine has a new template mechanism to make it easier to encode complex file templates. These templates map simply to `edit_line` bundles in the following way.

- Each line in a template maps to a separate `insert_lines` promise unless it is grouped with '`[%CFEngine BEGIN %]`' and '`[%CFEngine END %]`' tags.
- Each multi-line group, marked with '`[%CFEngine BEGIN %]`' and '`[%CFEngine END %]`' tags maps to a multi-line `insert_lines` promise, with `insert_type => "preserve_block"`.
- Each line that expresses a context-class: '`[%CFEngine` *classexpression*`:: %]`' maps to a normal class expression in a bundle.

The order of lines in the template is preserved within each block, or if `edit_defaults` is used to empty the resulting generated file before editing: e.g. with the standard library method:

```
  "/tmp/expander"

            create => "true",
     edit_template => "/home/a10004/input.dat",
     edit_defaults => empty;
```

Choosing an approach to file editing

There are two decisions to make when choosing how to manage file content:

How can the desired content be constructed from the necessary source(s)?
Is there more than one source of infromation that needs to be merged?

Do the contents need to be adapted to the specific environment?
Is there context-specific information in the file?

Use the simplest approach that requires the smallest number of promises to solve the problem.

Pitfalls to watch out for in file editing

File editing is different from most other kinds of configuration promise because it is fundamentally an order dependent configuration process. Files contain non-regular grammars. CFEngine attempts to simplify the problem by using models for the file structure, essentially factoring out as much of the context dependence as possible.

Order dependence increases the fragility of maintainence, so you should do what you can to minimize it.

- Try to use substitution within a known template if order is important.

The simplest kinds of files for configuration are line-based, with no special order. For such cases, simple line insertions are usually enough to configure files.

The increasing introduction of XML for configuration is a major headache for configuration management.

CFEngine®