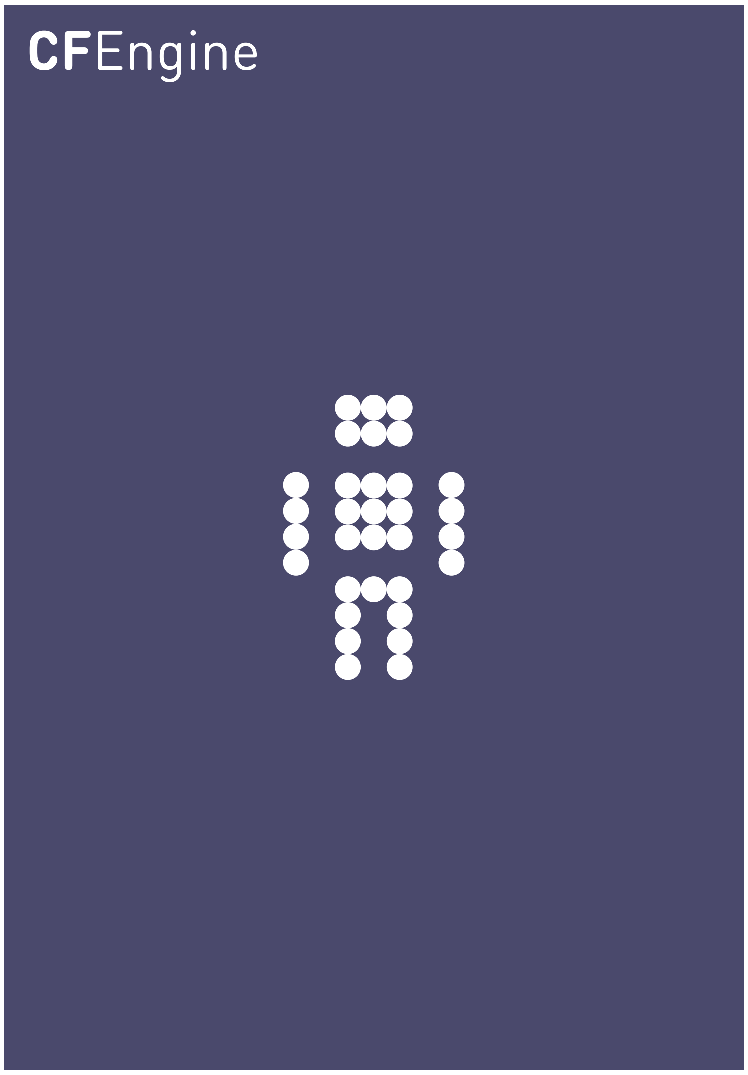

# Promising and Editing File Content

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

## What is convergent file editing?

Covergent file editing is the ability to specify the final state of a configuration file or document and have that result maintained over time, independently of its initial state. It is about keeping certain promises about the file's content, not on the process by which edits are made.

Cfengine allows you to model whole files or parts of files, in any format, and promise that these fragments will keep 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.

## 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 or certain approach. | Hard to specialize the result and the source must still be maintained by hand. |
| 2. Deterministic or certain approach. | 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.

## 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",

     # 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 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"
```

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

```
       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.

## 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.

## Examples of file editing

### Copying a 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 two approaches here:

1. Copy a template then edit it.
2. Copy a template with macro-substitution.

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");
}
```

For example, the file might look like this:

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

These variables will be filled in by CFEngine assuming they are defined. 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 => ifelapsed("60");
}

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

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

## Modifying the current state of a 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";
}
```