Getting Started with Cfengine 3 (Draft, 24 Aug 2011)

Contents

About this Document

Copyright

Purpose

Help new users come up to speed on Cfengine 3 by (a) providing practical examples, and (b) introducing concepts in logical sequence.

Intended Audience

You should have a basic understanding of Unix system administration.

What you will get out of this guide

You'll be able to install Cfengine and to start writing your own policies.

Contributions welcome

Contributions of practical examples of using Cfengine 3 to manage systems are most welcome. Please email the author at aleksey@verticalsysadmin.com.

Chapter 1

What is Cfengine?

1.1 What is Cfengine?

Cfengine is a change management system that improves the sysadmin's quality of life.

Cfengine helps system administrators by:

(a) Automating changes. Describe the change once, and Cfengine will implement it on your servers - even if you have tens of thousands of servers. A change that would take hours or days to roll out manually can be made automatically in minutes, and with more consistent results.

(b) Abstracting differences between various operating systems. For example, your policy can say "file share Reports on remote file server Fileserver1 should be mounted on the local directory /Reports" and you don't have to remember if your mount command is in /sbin/mount or /usr/bin/mount; or if your mount table is in /etc/fstab or /etc/vfstab.

Cfengine can handle over 32 different operating systems, including sun3, hpux, aix, sco, linux, various BSDs, cray, nextstep and qnx. Even Windows.

Simple sys admin tasks like checking the process table or monitoring percent of disk free wary between these OSes — Cfengine will take care of the variance, so you can focus on the actual policy you want to implement, rather than the details of that implementation.

(c) Documenting the intention of the configuration. Why is it configured that way? Who cares about it? What depends on it? What does it depend on? All this metadata can be embedded in the configuration policy so it is never lost or out of date in a separate document removed from reality.

(d) Continuously auditing the system and fixing deviations from policy, keeping the systems in order and operating smoothly and predictably.

(e) Cfengine is implemented as an agent-based system, which means each server is responsible for its own configuration, and any central servers are used only to help distribute policies or files which makes Cfengine very scalable (a single Cfengine server can handle hundreds of nodes straight out of the box; and with tuning, can handle thousands.)

Why use Cfengine?

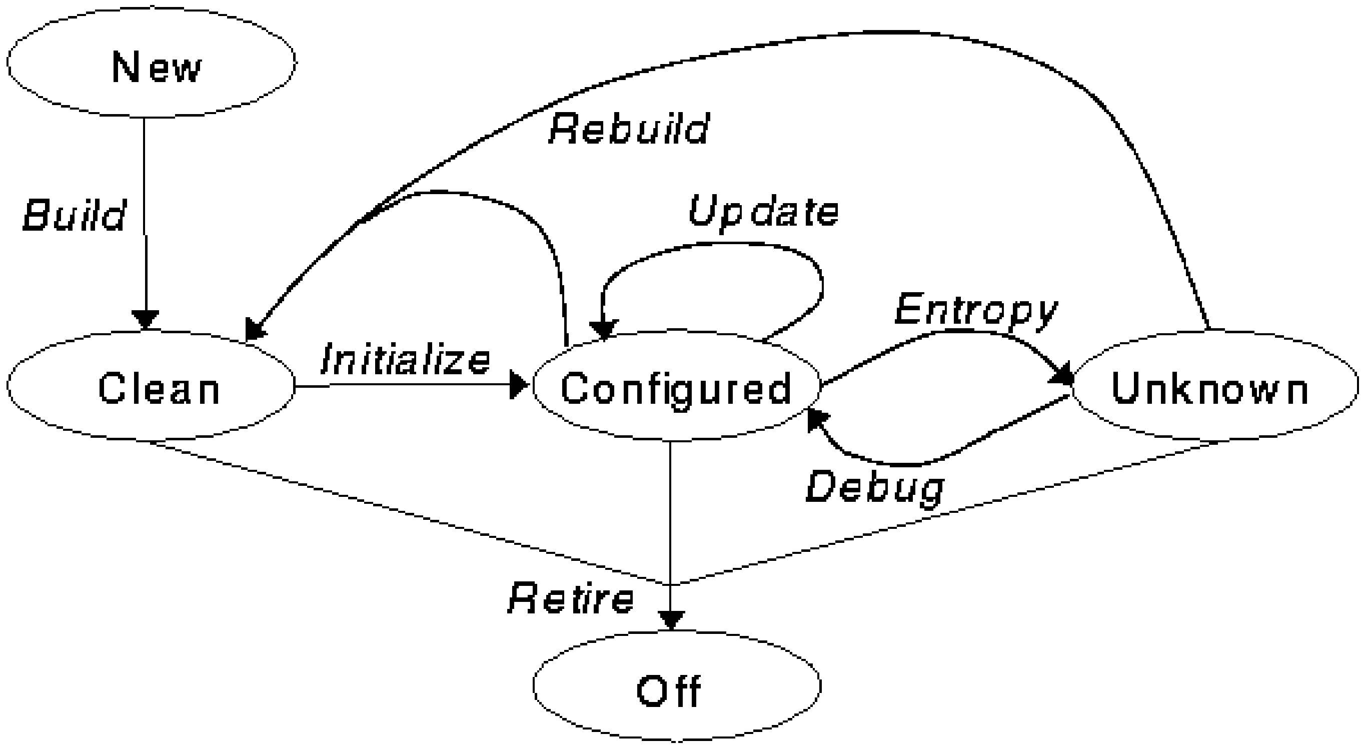
Cfengine automates a lot of the repetitive and not really intellectually stimulating work of managing servers, which allows system administrators to spend more time on the more interesting work of system engineering. Practically speaking, it changes the sysadmin to server ratio from tens/hundreds of servers per sysadmin, to thousands/tens of thousands of servers.

Cfengine enables scalability

Challenges to scalability (administering lots of servers) include: * State Drift (who made this change? why is this server configured like this? I thought we agreed to turn of FTP and use scp, why is FTP still running on this server? I know turned off all FTP servers last week.) * Change management (change httpd.conf on 50 servers) * Knowledge Management (who configured our postfix servers to reject mail from certain IP addresses? when? why? Or, why does our database allow connections from X network?)

Cfengine helps the sysadmin handle these challenges by automating change management and integrating it with knowledge management.

"The Lifecycle of a Machine" diagram from "An Analysis of UNIX System Configuration" by Remy Evard, 1997



Cfengine automates transition to the Configured state. It makes the Configured state the gravity pull point, to pull the machine into the Configured state.

1.2 Why choose Cfengine?

1.2.1 Small resource footprint

Cfengine is popular among system administrators because it is written in C and has very few dependencies, which keeps its resource footprint small and allows it to run in a wide variety of computing environments, from tiny embedded sensors to distributed supercomputers.

1.2.2 Backed by scientific research

Cfengine arose in synergy between academic research led by recognized visionary Mark Burgess (Earth's first Professor of System Administration) and use by industry, and continued communication and coordination between the two.

1.2.3 Proven track record

First released in 1993, Cfengine now manages over a million computers worldwide, and has a proven performance record. Designed with security in mind from the ground up, Cfengine has had only 3 major security issues in its 18 years of existence, which compares favorably even to security-related software such as OpenSSL or OpenSSH.

1.2.4 Widest range of supported operating systems

Cfengine, being written in C and having few dependcies, runs on the widest variety of operating systems, from crusty old to shiny new, and is portable, to boot.

1.3 How does Cfengine work?

Cfengine uses a powerful and flexible language for describing the intended system configuration. The Cfengine language is low-level (dealing with the nitty gritty details of system administration, such as file attributes and contents, packages, running processes and services) but is capable of high level abstraction, such as "webserver on", which takes care of installing the necessary packages, editing the Web server config files if necessary, configuring the OS to start the Web server at boot, configuring the host firewall to allow incoming TCP connections on ports 80 and 443, and starting the Web server daemon process.

How does the Cfengine language work?

Cfengine uses a flexible and powerful low-level language with knowledge management features that allow abstraction and summarization of low-level details into high-level concepts.

The Cfengine language describes the desired state. The Cfengine audits the system against that description and makes any changes necessary to bring it into that state. The guiding principle is convergence to desired state, also known as self-healing.

Chapter 2

Installing Cfengine

Source code can be downloaded from http://cfengine.com/source_code

Follow the quick start guide at http://cfengine.com/getting_started

Chapter 3

Basic Examples (Part 1)

3.1 Controlling a file: Disabling non-root logins

Example 3.1 Disabling non-root logins

0010_Basic_Examples._Create_a_file.cf

```
bundle agent example {

    files:

        "/etc/nologin"

            create  => "true",
            comment => "Prevent non-root users from logging into the system.";

}
```

This is a promise of type "files" (which means it deals with a file or files), the exact object being affected by the promise is "/etc/nologin", and the promise has two attributes: create (set to true, so the file will be autocreated if it isn't there), and a comment with is the purpose of the promise.

The promise is in a bundle, as it has to be to be addressable by the Cfengine agent, and the promise bundle has a type, "agent" (which means it is intended for the agent part of Cfengine, that's the part that makes changes to your system), and it has a name, "example".

To try the example, download the "practical_examples" tarball from www.verticalsysadmin.com/cfengine/practical_examples.tar and run cf-agent in Inform mode so you can see it create the file:

```
cf-agent -I -f ./001_Basic_Examples__create_a_file.cf -b example
```

COMMAND LINE ARGUMENTS

- -I Inform me of any changes to the system
- -f Specifies the policy File name
- -b Specifies the Bundlesequence (what promise bundles should be processed, and in what order)

Output

```
 >> Using command line specified bundlesequence
 -> Created file /etc/nologin, mode = 600
```

Note
Cfengine uses the → prefix to indicate changes it has made to the system.

Here are examples of handling processes nicely and not so nicely.

The first example specifies a command to run if there is an entry in the process table matching "cups":

3.2 Controlling a process: shutting down CUPS daemon

Example 3.2 Removing a process gracefully

0020_Basic_Examples._No_CUPSd_process._Stop_gracefully.cf

```
bundle agent example {

    processes:

        "cupsd"

            process_stop => "/etc/init.d/cups stop",
            comment => "We don’t want print services on our Web servers.";
}
```

3.3 Blacklisting a process: shoot IRC bots on sight

In the second example, any process matching "eggdrop" (an IRC bot) promises to experience the signals TERM and then KILL.

Example 3.3 Terminating a process with prejudice

0030_Basic_Examples._No_eggdrop_process._Terminate_with_prejudice.cf

```
bundle agent example {

    processes:

        "eggdrop"

            signals => { "term", "kill" },
            comment => "We don’t want IRC bots on our Web servers.";

}
```

There are many safety mechanisms in Cfengine. It won’t signal itself, even though the process table scan matches on "eggdrop". Verbose run of the above example (with -v) shows:

```
cf3> Observe process table with /bin/ps -eo
user,pid,ppid,pgid,pcpu,pmem,vsz,pri,rss,nlwp,stime,time,args
cf3>
cf3>     .....................................................
cf3>     Promise handle:
cf3>     Promise made by: eggdrop
cf3>
cf3>     Comment:  We don’t want IRC bots on our Web servers.
cf3>     .....................................................
cf3>
```

```
cf3>  ->  Found matching pid 30324
     (root     30324 30133 30324  0.0  0.0  38364  21  3824    1 07:50
00:00:00 cf-agent -f ./practical_examples/0030_Basic_Examples._No_eggdrop_process. ↩
   _Terminate_with
_prejudice.cf -v -b example -K)
cf3>  !! cf-agent will not signal itself!
cf3>  -> No restart promised for eggdrop
cf3>
```

Chapter 4

Promise Theory

4.1 What is Promise Theory?

Promise Theory is a model of system orchestration created to understand the Cfengine and other distributed systems, and understand their predictability.

You do not need to know anything about Promise Theory to use Cfengine. Just think of Cfengine as a descriptive language, with a simple but flexible syntax.

Promises represent an expression of intention. Their value is in communicating desired outcomes or states.

The basic elements of a promise are:

a → b

(a) is the promiser

(b) is the promisee

Between them is the promise itself, including its nature and any constraints on the promise.

Example 4.1 Real world example

The author of this document promises to help you get started with Cfengine.
Promiser: the Author.
Promisee: the Reader.
Promise: the Reader will be able to start using Cfengine after Reading and Practicing.

Example 4.2 System administration example

Google promises to the public that www.google.com website will be up and there will be a useful search engine there.

4.2 The Basic Form of a Cfengine promise

In the Cfengine language, a promise has a type (which declares the area it deals with: for example, files or processes) and it has constraints which are expressed as attribute/value pairs (such as whether to autocreate a promised file, or what signal to send to a process).

Cfengine promises have the following form

```
promisetype:

    "promiser"  ->  promisee,
        attribute1,
        attribute2,
        ...
        attributeN;
```

Promisees are optional. They are used in the commercial versions of Cfengine for knowledge management and impact analysis.

The promisee, if present, is followed by a comma; and promise attributes are separated by commas.

The promise is terminated by a semicolon.

Chapter 5

Types of promises in the Cfengine language

What kind of promises can be made in the Cfengine language?

Here is a starting set:

files A promise about a file, including its existence, attributes and contents.

packages A promise to install (or remove or update or verify) a package.

processes A promise concerning items in the system process table (such as promising that a process is present or absent).

vars A promise to be a variable, representing a value.

reports A promise to report a message.

commands A promise to execute a command.

methods A promise to take on a whole bundle of other promises.

storage A promise to handle storage (such as: to make sure a filesystem is mounted, check disk usage on a filesystem, etc.)

And more.

In the commercial edition of Cfengine, promises can be made concerning databases, including Windows registry contents, concerning services, and more yet.

Chapter 6

Definition: Promise Body

Promise body A promise body is a collection of promise attributes that detail and constrain the nature of the promise.

Think "body" as in the body of a letter, or the body of a contract - it's where the details are.

The form of a promise body

```
promise_type:                         # this is the promise type (e.g. files,
                                      # processes, commands, etc.)

    "promiser"                        # this is the promiser, the part of
                                      # the system that will be affected by
                                      # the promise.


         attribute_1 => value_1,      # This is the promise BODY.
         attribute_2 => value_2,      #   The promise body details and
         ...                          #   constrains the nature of the
         attribute_n => value_n;      #   the promise.  It consists of
                                      #   attributes which have values.
```

Example 6.1 A promise with a body comprised of 3 attributes: create the file, update the timestamp, and metadata about intention of the promise.

0055_Basic_Examples._Touch_a_file.cf

```
bundle agent example {

    files:

        "/var/cfengine/i_am_alive"

            create  => "true",   # promise body starts here
            touch   => "true",   # promise body
            comment => "Update timestamp on this file so my sysadmin knows I am alive and  ↩
                faithful.";    # promise body ends here
}
```

Output

```
# cf-agent -f ./0055_Basic_Examples._Touch_a_file.cf  -I -b example
 >> Using command line specified bundlesequence
 -> Created file /var/cfengine/i_am_alive, mode = 600
 -> Touched (updated time stamps) /var/cfengine/i_am_alive
# cf-agent -f ./0055_Basic_Examples._Touch_a_file.cf  -I -b example -K
 >> Using command line specified bundlesequence
 -> Touched (updated time stamps) /var/cfengine/i_am_alive
 -> Touched (updated time stamps) /var/cfengine/i_am_alive
#
```

Note

You might have noticed that on the second run, cf-agent touched the file twice. This is because cf-agent may do up to three passes over its policy each time it runs. This is part of its convergent operation and is how Cfengine works. In this case the result is the same whether it does one pass or two. Cfengine is convergent toward ideal state. Like a doctor, its motto is "do no harm". It always tries to move the system closer to desired state, never away from it. If you look at the first run, during the first pass it was unable to keep the promise to update the timestamp on the file because the file did not exist. So it created the file during the first pass, and touched the timestamp during the second.

Chapter 7

Definition: Promise Bundle

Promise bundle A promise bundle is a group of one or more promises. It allows us to group related promises, and to refer to such groups by name. It's one of the basic building blocks of configuration in Cfengine. It allows us to store policies so that Cfengine can retrieve and execute them.

Note
Identifiers must be made of alphanumeric characters and underscores only: A-Z a-z 0-9 _

Example 7.1 A promise bundle containing one promise:
0010_Basic_Examples._Create_a_file.cf

```
bundle agent example {

    files:

        "/etc/nologin"

            create  => "true",
            comment => "Prevent non-root users from logging into the system.";

}
```

Squiggly brackets open and close the bundle.

Multiple promises in a bundle must be separated by semicolons. Every complete statement in Cfengine is terminated by a semicolon.

Example 7.2 A bundle with multiple promises

0040_Basic_Examples._Two_promises_in_one_bundle.cf

```
bundle agent example {

    files:

        "/tmp/hello"

            create  => "true";

    files:

        "/tmp/world"

            create  => "true";

}
```

A promise bundle must have a type. We start by looking at "agent" type bundles, which are promises handled by cf-agent, which is the part of cfengine that actually makes changes to the system. (You could say cf-agent is the "workhorse" part of Cfengine.) There are other types of bundles specific to other components of Cfengine, such as "server" (policy and file distribution server).

Chapter 8

Gotcha! Cfengine locks. The -K switch.

Let's say you run a policy that creates a file, as with our first example, /etc/nologin.

Then you immediately remove that file and run the above example again. You might expect it to re-create /etc/nologin, but it will do nothing. But if you wait a minute, and run it again, it will re-create the file.

Be aware of this, otherwise you'll end up wondering why your configuration "isn't working".

This is a safeguard against Cfengine running amok with infinite loops chewing up resources. Cfengine monitors when it last took an action, and won't repeat it until the default period (1 minute) elapses.

However, you can force immediate execution with the -K switch to cf-agent.

Example 8.1 Forcing immediate execution

```
# cf-agent -I -b example -f ./0010_Basic_Examples._Create_a_file.cf    ❶
 >> Using command line specified bundlesequence
 -> Created file /etc/nologin, mode = 600
# rm /etc/nologin                                                     ❷
# cf-agent -I -b example -f ./0010_Basic_Examples._Create_a_file.cf    ❸
 >> Using command line specified bundlesequence
# cf-agent -I -b example -f ./0010_Basic_Examples._Create_a_file.cf -v | \
  grep -i "lock.*elapsed" ❹
cf3>  XX Nothing promised here [lock.example.files.create.comment.-ve._e] (0/1 minutes  ↩
    elapsed)
# cf-agent -I -b example -f ./0010_Basic_Examples._Create_a_file.cf -K ❺
 >> Using command line specified bundlesequence
 -> Created file /etc/nologin, mode = 600
#
```

❶ Use cf-agent to make a change to our system.

❷ Clean the slate, so that cf-agent has something to do when it runs next.

❸ Run cf-agent again. We expect cf-agent to create the file again, but it does nothing.

❹ Verbose mode shows us that no action was taken because less than 1 minute elapsed.

❺ Use -K switch to force immediate execution, bypassing the lock subsystem.

Chapter 9

Basic Examples (Part 2)

Here are some more examples for you to try. In each example, identify the promise bundle and the promise body.

9.1 Setting and Using Scalar Variables

Example 9.1 Setting and Using Scalar Variables

0130_Basic_Examples._Be_a_variable.cf

```
bundle agent example {

    vars:

        "my_string"

            string  => "The answer is ...";

    vars:

        "my_int"

            int     => "42";


    commands:

        "/bin/echo My string: $(my_string).  My int: $(my_int)";

}
```

Output

```
# cf-agent -b example -KIf ./0130_Basic_Examples._Be_a_variable.cf
 >> Using command line specified bundlesequence
 -> Executing '/bin/echo My string: The answer is ....  My int: 42' ...(timeout=-678,owner ←
     =-1,group=-1)
Q: ".../bin/echo My st": My string: The answer is .... My int: 42
I: Last 1 quoted lines were generated by promiser "/bin/echo My string: The answer is ....  ←
     My int: 42"
 -> Completed execution of /bin/echo My string: The answer is ....  My int: 42
#
```

9.2 Setting and Using List Variables

Setting and Using List Variables - Implicit Looping Over List Elements

0550_Data_Types._List_variables_and_implicit_looping.cf

```
bundle agent example {

vars:

   "shopping_list"      slist   => {
                                       "apples",
                                       "oranges",
                                       "bananas",
                                       "grapes",
                                       "plantains",
                                  };
############################################################

commands:

        "/bin/echo $(shopping_list)";
}
```

Cfengine will loop over the list, generating a commands promise for each list element

```
# cf-agent -f ./0550_Data_Types._List_variables_and_implicit_looping.cf -b example
Q: ".../bin/echo apple": apples
I: Last 1 quoted lines were generated by promiser "/bin/echo apples"
Q: ".../bin/echo orang": oranges
I: Last 1 quoted lines were generated by promiser "/bin/echo oranges"
Q: ".../bin/echo banan": bananas
I: Last 1 quoted lines were generated by promiser "/bin/echo bananas"
Q: ".../bin/echo grape": grapes
I: Last 1 quoted lines were generated by promiser "/bin/echo grapes"
Q: ".../bin/echo plant": plantains
I: Last 1 quoted lines were generated by promiser "/bin/echo plantains"
#
```

9.3 Creating a File (Using Knowledge Management Features)

Example 9.2 0150_Basic_Examples._Create_a_file,_with_KM_features.cf

```
bundle agent example {

files:

   "/tmp/testfile" -> "Application X",
                      # document which application relies on
                      # or uses this file

        handle => "create_testfile", # a name for this promise.
                                     # can be used with depends_on
                                     # attribute in another promise
                                     # to document dependency

        comment => "/tmp/testfile must be mode 740 for application X to work",
                   # this comment will show up in verbose or debug modes

        create  => "true";
}
```

Chapter 10

Editing Files (Part 1)

10.1 Promising a line of text in a file

Example 10.1 Make sure a file contains a line of text.

0160_Editing_Files._Edit_a_file.cf

```
bundle agent example {

files:

   "/tmp/testfile"

        create  => "true",
        edit_line => proper_greetings;
}


####################################################


bundle edit_line  proper_greetings {

insert_lines:

    "Good morning, how are you today?";

}
```

10.2 Promising the entire contents of a file

Example 10.2 Make sure a file contains ONLY that line of text.

0165_Editing_Files._Edit_a_file,_strict_match.cf

```
bundle agent example {

files:

   "/tmp/testfile"

        create  => "true",
        edit_line => proper_greetings;
}


bundle edit_line  proper_greetings {


delete_lines:

    ".*";


insert_lines:

    "Good morning, how are you today?";


}
```

10.3 Adding a user account group

Example 10.3 Make sure /etc/group contains an entry for "cfengine" group (GID 502)

0170_Editing_Files._Edit_a_file,_add_a_group.cf

```
# Make sure /etc/group contains an entry for
# a "cfengine" group, GID 502

bundle agent example {

    files:

        "/etc/group"

            edit_line => cfengine_group;

}


bundle edit_line cfengine_group {

    insert_lines:

        "cfengine:x:502:";

}
```

10.4 Removing a user account group

Example 10.4 Make sure /etc/group does NOT contain an entry for the "games" group.

0175_Editing_Files._Removing_the_games_group_from_etc_group_file.cf

```
bundle agent example {

    files:

        "/etc/group"

            edit_line => delete_group("games:x:[0-9]+:");
                                     # note the parameter
                                     # you can parameterize bundles
}


bundle edit_line delete_group(group) {

    delete_lines:
        "$(group).*";
}
```

Output

```
# grep games /etc/group
games::20:
# cf-agent -f ./0175_Editing_Files._Removing_the_games_group_from_etc_group_file.cf -KI -b  ↩
    example
 >> Using command line specified bundlesequence
 -> Edited file /etc/group
# grep games /etc/group
#
```

Chapter 11

Notes on Syntax (Part 1)

11.1 The promise type can be implicit.

If unspecified, the previous promise's type is assumed. This is (an optional) scalability feature.

Promise types can be implicit. If you don't specify a promise type, Cfengine will reuse the promise type of the prior promise.

This is a scalability feature.

It's a type of compression, to allow quicker reading of the policy, so that the intention shines through. (Less text, more meaning.)

Example 11.1 Implicit promise typing allows us to condense policy without losing meaning.

0045_Basic_Examples._Two_promises_in_one_bundle._Condensed.cf

```
bundle agent example {

    files:

        "/tmp/hello" create  => "true";
        "/tmp/world" create  => "true";
}
```

Example 11.2 Take advantage of implicit promise typing to compact your code

0195_Notes_on_Syntax._Be_a_variable._Promise_type_can_be_implicit.cf

```
bundle agent example {

    vars:

        "my_string" string  => "The answer is...";

        "my_int" int      => "42";

        "my_real" real    => "3.14159";
}
```

11.2 Whitespace/Indentation Do Not Matter.

White space and indentation do not matter in Cfengine. Observe the following:

Example 11.3 Setting and Using Scalar Variables (No Whitespace or Indentation)

0190_Notes_on_Syntax._Be_a_variable._Whitespace_and_indentation_do_not_matter.cf

```
# Indentation does not matter

bundle agent example {

vars: "my_string" string  => "String contents...";
      "my_int"    int     => "42";
      "my_real"   real    => "3.14159";

reports: "$(my_string) $(my_int) $(my_real)";

}

# Whitespace does not matter.  All one line:

bundle agent example2 { vars: "my_string" string  => "String contents..."; "my_int" int  ←
        => "42"; "my_real" real    => "3.14159"; reports: "$(my_string) $(my_int) $(my_real) ←
    "; }
```

Output

```
# cf-agent -f ./0150_Basic_Examples._Be_a_variable. ←
    _Whitespace_and_indentation_do_not_matter.cf -b example -K
Q: ".../bin/echo The a": The answer is... 42 3.141590
I: Last 1 quoted lines were generated by promiser "/bin/echo The answer is...
42 3.141590"
#
```

11.3 Identifier Naming

Identifiers must be made of alphanumeric characters and underscores only: A-Z a-z 0-9 _

Valid identifiers: hello, Text_String, counter01

Invalid identifiers: my-identifier, $hello, Test!, @#$%$

This applies to all identifiers: bundle names, variable names, promise handles, etc.

Chapter 12

Providing Context to Promises (Part 1)

12.1 Definition: Class

Cfengine allows you to specify the context of a promise, that is to say, when and where that promise should be active. "Class" in this case refers to groups, sets, categories of hosts. It's just the English word class meaning group or category, nothing to do with object oriented programming.

Class "A set or category of things having some property or attribute in common and differentiated from others by kind, type, or quality." Compact Oxford Dictionary

Class "A class of things is a group of them with similar characteristics. . . . the largest class of racing sailboats in the world." Collins COBUILD Student Dictionary

Cfengine classifies its environment each time it wakes up to decide which promises apply "here".

This way you can write promises that only apply to Solaris servers (for example, configuring /etc/vfstab) or that are only active on weekends between 2 AM and 5 AM (kicking off backups or doing file system file integrity checks (a la "tripwire").

The form of a Cfengine promise (shows class)

```
promise_type:

    class::             # where/when the promise is active,
                        # (the context of the promise)
        "promiser"

            attribute1 => value1,
            attribute2 => value2,
            ...
            attributeN => valueN;
```

12.2 Determining OS type

Example 12.1 Determining OS type

0200_Patterns._Classes._Hello_world_report_of_OS_type.cf

```
bundle agent example {

    reports:

        WinXP:: "Hello world! I am running on a Windows system.";

        linux:: "Hello world! I am running on a Linux system.";

        redhat:: "Hello world! I am running on a redhat Linux system.";
}
```

Output on a CentOS system (which is recognized as a Red Hat derivative)

```
# cf-agent -f ./0200_Patterns._Classes._Hello_world_report_of_OS_type.cf -b example
R: Hello world! I am running on a Linux system.
R: Hello world! I am running on a redhat Linux system.
#
```

Note
Cfengine prefixes the output of reports type promises with "R: ".

Class expressions are optional; if you don’t specify one in your first promise, the class will default to "any" (which is a special class that is always true). If you don’t specify a class in your second promise, it will re-use the class from the first promise.

Warning
If you don’t want your promise to be constrained to the preceeding promise’s class, explicitly specify the class context of your promise (if the class is "any", then say "any").

12.3 Reporting the day of the week

Example 12.2 Reporting the day of the week

0210_Patterns._Classes._Report_day_of_the_week.cf

```
bundle agent example {

    reports: Monday::    "Hello world! I love Mondays!";
    reports: Tuesday::   "Hello world! I love Tuesdays!";
    reports: Wednesday:: "Hello world! I love Wednesdays!";
    reports: Thursday::  "Hello world! I love Thursdays!";
    reports: Friday::    "Hello world! I love Fridays!";

    reports: Saturday::  "Hello world! I love weekends!";
    reports: Sunday::    "Hello world! I love weekends!";
}
```

Let’s run this on a Sunday

```
# date
Sun Aug  7 14:29:10 PDT 2011
# cf-agent -f ./0210_Patterns._Classes._Report_day_of_the_week.cf -b example
R: Hello world! I love weekends!
#
```

12.4 Using Network Address Classes to Guess Roles

Example 12.3 Figure out if I am a database server (backend) or a Web server (frontend)

0240_Patterns._Classes._Using_classes_to_determine_role.cf

```
bundle agent example {

    reports:
      ipv4_205_186_156:: "I am on our public net. I’ll be a Web server.";

    reports:
      ipv4_10:: "I am on our private net. I’ll be a database server.";

}
```

12.5 Built-in classes

You can check how Cfengine classifies your system by running cf-agent in verbose mode (cf-agent -v -f . . .) and checking the "Defined classes" section.

You’ll see classes like:

12.5.1 Time-related classes

- A day of the week (in the form Monday, Tuesday, Wednesday, ..).
- An hour of the day, in the current time zone (in the form Hr00, Hr01 . . . Hr23).

- An hour of the day GMT (in the form GMT_Hr00, GMT_Hr01 ... GMT_Hr23). This is consistent the world over, in case you need virtual simulteneity of change coordination.
- Minutes in the hour (in the form Min00, Min17 ... Min45).
- A five minute interval in the hour (in the form Min00_05, Min05_10 ... Min55_00).
- A fifteen minute (quarter-hour) interval (in the form Q1, Q2, Q3, Q4).
- An expression of the current quarter hour (in the form Hr12_Q3).
- A day of the month (in the form Day1, Day2, ... Day31).
- A month (in the form January, February, ... December).
- A year (in the form Yr1997, Yr2004).
- A shift in Night,Morning,Afternoon,Evening, which fall into six hour blocks starting at 00:00 hours.
- A "lifecycle index", which is the year number modulo 3 (in the form Lcycle_0, Lcycle_1, Lcycle_2, used in long term resource memory).

12.5.2 Classes based on OS characteristics and settings

- The name of an operating system architecture e.g. ultrix, sun4, etc.
- The unqualified name of a particular host (e.g., www). If your system returns a fully qualified domain name for your host (e.g., www.iu.hio.no), cfengine will also define a class for the fully qualified name, as well as the partially-qualified component names iu.hio.no, hio.no, and no.
- The IP address octets of any active interface (in the form ipv4_192_0_0_1, ipv4_192_0_0, ipv4_192_0, ipv4_192).
- The names of the active interfaces (in the form net_iface_xl0, net_iface_vr0).
- System environmental information reported by cf-monitord (such as CPU utilization and network utilization entropy (unusually high/low on a per-port or per-protocol basis))
- On Solaris-10 systems, the zone name (in the form zone_global, zone_foo, zone_baz).

12.5.3 Classes based on the version of Cfengine

- The version of Cfengine (in fully and partially qualified forms: cfengine_3 cfengine_3_1 cfengine_3_1_5)
- The edition of Cfengine (community or enterprise)
- Where was it built (e.g. compiled_on_linux_gnu)
- How Cfengine is being run (e.g. verbose_mode)

And more.

Chapter 13

Providing Context to Promises (Part 2)

13.1 Definition: Class Expression

One can provide the context for a promise using a class expression rather than a single class. This is part of what makes Cfengine so powerful and flexible.

Classes are combined with:

```
NOT: !
AND: . (dot) or &
OR: |
Parentheses ()
```

Example 13.1 Simple Example of combining classes:

```
(freebsd&Hr03&Sunday)
```

Example 13.2 Convoluted Example of combining classes:

```
(freebsd&(!Hr02))|(solaris&Hr03)
```

The form of a Cfengine promise

```
promise_type:

    class_expression::      # logical expression combining
                            # two or more clases

        "promiser"

            attribute1 => value1,
            attribute2 => value2,
            ...
            attributeN => valueN;
```

13.2 Setting Context to "Linux servers AND between 15:00 and 15:59"

Example 13.3 Example - class expression - combining time and OS type

0220_Patterns._Classes._OS_and_time_expression.cf

```
bundle agent example {

    reports:

      linux&Hr15:: "Linux system AND we are in the 15th hour.";

      linux&Hr22:: "Linux system AND we are in the 22nd hour.";
}
```

Output

```
# date
Sun Aug  7 15:14:13 PDT 2011
# cf-agent -f ./0220_Patterns._Classes._OS_and_time_expression.cf -b example
R: Linux system AND we are in the 15th hour.
#
```

13.3 Setting Context to "Redhat servers on Thursdays OR Windows servers on Wednesday"

Example 13.4 A made up example of an expression of expressions

0230_Patterns._Classes._Class_expression.OS_and_time.cf

```
bundle agent example {

    reports:

      (redhat&Thursday)|(windows&Wednesday)::

        "This promise will execute on Redhat servers on Thursdays; or on Windows servers on ↩
             Wednesdays";
}
```

Chapter 14

Promise Body-Parts

14.1 Definition: Promise Body-Parts

The basic building blocks of Cfengine configuration are bundles and body-parts.

Promise bundle A promise bundle is a collection of promises under a single name.

Promise bundles have the following form

```
bundle bundle-type identifier
{
...(promises)
}
```

Example 14.1 An example cf-agent promise bundle called "example"

```
bundle agent example {
    ...(promises)
}
```

Promise body A promise body is a collection of promise **attributes** for a single promise. These attributes detail and constrain the nature of that promise.

Promise bodies have the following form

```
bundle agent example {

promise_type:

    "promiser"

         attribute_1 => value_1,       # This is the promise BODY.
         attribute_2 => value_2,       #   The promise body details
         ...                           #   and constrains the nature
         attribute_n => value_n;       #   of the promise.

}
```

Example 14.2 An example cf-agent promise bundle called "example" that contains a body comprised of 3 attributes: create the file, update the timestamp, and metadata about intention of the promise.

0055_Basic_Examples._Touch_a_file.cf

```
bundle agent example {

    files:

        "/var/cfengine/i_am_alive"

            create  => "true",    # promise body starts here
            touch   => "true",    # promise body
            comment => "Update timestamp on this file so my sysadmin knows I am alive and  ←
                faithful.";     # promise body ends here
}
```

Promise body-part A collection of promise body constraints which further qualify and constrain the nature of a promise. Abstracted from the promise and parameterizable, promise body-parts allow us to express high level concepts by hiding the low-devel details within the body-part. Because the body-part is separate from the promise bundle, it can be re-used (used by other promise bundles).

The body-parts block has the form:

```
body constraint_type body_part_identifier {

    attribute1 => value1;  # (attributes/constraints)
    attribute2 => value2;
    ...
    attributeN => valueN;
}
```

A body-part may have parameters.

What connects the bundle block and the body-part block is a reference inside a promise of the form *constraint_type ⇒ body_part_identifie* which means here is a promise constraint of type "constraint_type" but the details of that constraint are stored in a body-part named "body_part_identifier".

A body-part expands on a promise. Because it is a separate piece, it can be considered an attachment to a promise.

The body-part is outside the bundle and is re-usable by other bundles.

Chapter 15

GOTCHA! Body-Part terminology

Important
Body-Part is also referred as Body, Compound Body, Compound Body Part, External Body Part, Body Part Template. Please make sure you differentiate Body and Body-Part conceptually and don’t be blindsided when the word "body" is used to refer to either of them.

Chapter 16

Body-Part Examples

16.1 Setting File Attributes using a Body-Part

Example 16.1 0380_Body-Part_Examples._No_world_write_bit.cf

0380_Body-Part_Examples._No_world_write_bit.cf

```
bundle agent example {

files:

   "/tmp/testfile"

        comment => "/tmp/testfile must not be world-wirtable",
        perms   => no_world_writable_files_allowed;

}


#######################################################

body perms no_world_writable_files_allowed
{
mode   => "o-w";
}
```

16.2 Setting File Attributes using a Parameterized Body-Part

Example 16.2 Setting File Attributes using a Parameterized Body-Part

1020_COPBL._File_exists_and_is_mode_6_1_2_mog._Without_COPBL.cf

```
bundle agent example {

files:

   "/tmp/testfile"

        comment => "/tmp/testfile must be mode 612 for application X to work; it must be  ←
            owned by user aleksey and group cfengine",
        create  => "true",
        perms   => mog("612","aleksey","cfengine");

files:
   "/tmp/file2"

        create => "true",
        perms => mog("700", "root", "root");

}

############################################################

body perms mog(mode,user,group)
{
owners => { "$(user)" };
groups => { "$(group)" };
mode   => "$(mode)";
}
```

16.3 Setting File Attributes using a Parameterized Body-Part and Classes

Example 16.3 0383_Body-Parts_Examples._Setting_group_ownership_based_on_OS.cf

0383_Body-Parts_Examples._Setting_group_ownership_based_on_OS.cf

```
# Two bundles sharing a body-part that automagically sets the correct group ownership based ↩
     on OS

bundle agent example1 {

files:

   "/tmp/testfile"
        create  => "true",
        perms   => set_attributes("aleksey");
}


################################################

bundle agent example2 {

files:

   "/tmp/testfile2"
        create  => "true",
        perms   => set_attributes("rob");
}


################################################


body perms set_attributes(owner) {

    mode   => "0700";

    owners => { "$(owner)" };

  linux::  groups => { "wheel" };
  darwin:: groups => { "admin" };


}
```

16.4 Installing a Package

Example 16.4 Installing a Package

0381_Body-Part_Examples._Install_a_package.cf

```
bundle agent example {

packages:

  "php-mysql"

     package_policy => "add",   # Ensure that a package is present
     package_method => my_yum_method;
}

############################################################


body package_method my_yum_method {

     package_changes => "bulk"; # Can be "individual" or "bulk"

     package_list_command => "/usr/bin/yum list installed";
                             # command to run to get a list of packages

     package_list_update_ifelapsed => "1";  # 1 minute - default is 4 hours
                                            # this is how long cfengine will
                                            # cache the package list

     package_list_name_regex    => "([^.]+).*";            # Regexes needed
     package_list_version_regex => "[^\s]\s+([^\s]+).*";  # to parse the
     package_list_arch_regex    => "[^.]+\.([^\s]+).*";   # package list
     package_installed_regex => ".*installed.*";          # command output

     package_add_command => "/usr/bin/yum -y install";
                            # command to run to install a package
     package_delete_command => "/bin/rpm -e";
                            # command to run to delete a package
}
```

Output

```
# rpm -q php-mysql
package php-mysql is not installed
[cfengine00  practical_examples]# cf-agent -f ./0381_Body-Part_Examples._Install_a_package. ←
    cf -b example
# rpm -q php-mysql
php-mysql-5.1.6-27.el5_5.3
#
```

16.5 Removing a Package

Example 16.5 Removing a Package

0382_Body-Part_Examples._Remove_a_package.cf

```
bundle agent example {

packages:

  "php-mysql"

     package_policy => "delete",  # Ensure the package is not installed
     package_method => my_yum_method;
}

body package_method my_yum_method {

     package_changes => "bulk"; # Can be "individual" or "bulk"

     package_list_command => "/usr/bin/yum list installed";
                             # command to run to get a list of packages

     package_list_update_ifelapsed => "1";  # 1 minute - default is 4 hours
                                            # this is how long cfengine will
                                            # cache the package list

     package_list_name_regex    => "([^.]+).*";            # Regexes needed
     package_list_version_regex => "[^\s]\s+([^\s]+).*";  # to parse the
     package_list_arch_regex    => "[^.]+\.([^\s]+).*";   # package list
     package_installed_regex => ".*installed.*";          # command output

     package_add_command => "/usr/bin/yum -y install";
                            # command to run to install a package
     package_delete_command => "/bin/rpm -e";
                            # command to run to delete a package
}
```

Output

```
[cfengine00  practical_examples]# cf-agent -f ./0382_Body-Part_Examples._Remove_a_package. ↩
    cf -I -b example
 >> Using command line specified bundlesequence
Q:   rpm -e php-mysql  ...:
[cfengine00  practical_examples]#
```

Chapter 17

Body-Parts Standard Library

cfengine_stdlib.cf is a collection of body-parts useful for system administration

Chapter 18

Notes on Syntax (Part 2)

The basic Cfengine syntax pattern to describe configuration is: left-hand side ⇒ right-hand side.

a. Promise attributes
b. Example of user defined body on RHS
c. Example of builtin function on RHS
d. Example of scalar on RHS
e. Example of list on RHS

Chapter 19

The Fine Reference Manual

To start, read the first few chapters and get familiar with the "agent" section of the manual.

Chapter 20

How to have Cfengine automatically administer your test system

1. Populate /var/cfengine/inputs:

```
cp /usr/local/share/cfengine/masterfiles/*cf /var/cfengine/inputs
```

1. Put your local policy files into /var/cfengine/inputs

For example:

```
cp /home/user/test.cf /var/cfengine/inputs
```

1. Edit /var/cfengine/inputs/promises.cf 3.1. Add your policy files to inputs. 3.2. Add the names of your promise bundles to bundlesequence
2. Do a test run with "/var/cfengine/bin/cf-agent" and check if your policies are taking effect. Cfengine should bootstrap itself to run from cron when run for the first time.

Now, manually alter your system state — Cfengine should converge it back to desired state. Observe and confirm this for yourself.

That’s as far as the 3 hour "Getting Started with CFEngine 3" course goes.

Chapter 21

Professional Training and Implementation Consulting

Professional training in CFEngine is available. Please visit http://cfengine-book.eventbrite.com/

Implementation consulting is available. Please email aleksey@verticalsysadmin.com

Chapter 22

Outline for the rest of the training

Part II

1. Data Types
   a. Scalars and Lists
   b. Scalars
2. Simple examples
3. Special suffixes for Integer constants, complete with demo of "k" versus "K" (1000 vs. 1024)
   a. Lists
4. Simple example of list of strings.
5. Lists and implicit looping - another way to reduce noise and increase signal.
6. Concatenation of slists.
   a. Simple example of list of integers
   b. Simple example of list of real numbers
   c. Demonstration of typing: error message in trying to put a real into an integer.
7. Data Structures
   a. Arrays
   b. Example: an Array of Strings

Part III. Cfengine on your Server

1. Software Components
   a. Parts for validating and running policies: cf-promises, cf-agent, cf-execd
   b. Network file/policy server: cf-serverd
   c. Tools for Cfengine networking: cf-key and cf-runagent
   d. Other components: cf-monitord, cf-know, cf-report, cf-hub
2. The starting set of config files: what does what.

Part IV. Cfengine Language: the standard library, COBPL (Cfengine Open Promise

1. Community Open Promise Body Library, an introduction
2. Example: Package add using COPBL
3. Example: Creating a file and setting permissions
4. Example: Context sensitive file editing: set robs password
5. Example: Removing a file: Remove Apache httpd "welcome" page to hide our OS and httpd version.
6. Example: Making welcome.conf match copy from remote master

Part IV. Cfengine Language: Patterns + Promises = Configuration

1. The power of patterns - how a pattern increases efficiency, and changes the signal/noise ratio.
2. Lists
    a. implicit looping over a list of packages
    b. implicit looping over a list of files
3. Regular Expressions
    a. Example with files
    b. Example with processes
4. Body templates - patterns of promise attributes. Ability to parameterize promise attributes. Code re-use.
5. Classes - ("patterns in time and space") part 2
    a. Example: report type of weekday (uses ifvarclass, which allows to use variables in class expressions)
    b. Combining classes and functions: OR'ing of classes and fileexists function
    c. default classes from cf-agent
    d. default classes from cf-monitord
    e. Setting a private class based on hard classes expression
    f. Combining classes
    g. Example: configure iptables based on a class expression
    h. Example: Setting a custom class based on built-in classes - simple
    i. Example: Setting a custom class based on built-in classes - more complex
    j. Example: Setting a custom class based on function result
    k. Example: Determining role based on IP network

    ```
    5b. Classes - part 3 - Global vs local classes (scope)
    ```
6. Bundling and parameterizing promises with "methods" type promise
    a. "methods", a special promise type that lets promises call other promises
    b. a simple example
    c. on benefits of encapsulation and re-use

Part V. Practical Examples

1. File editing
    a. Embedding RCS Id tag
    b. Embedding SVN version tag

    c. Resolver configuration
    d. Exclude postgresql packages from CentOS Base YUM repository.
    e. Configure autofs to use LDAP
    f. Configure autofs using a template
    g. Make sure group exists and has correct GID

2. File copying

    a. Local copy (no cf-serverd required) - mirroring a local directory
    b. Remote copy - mirroring a remote directory
    c. Remote copy - with server round robin

3. Security

    a. Change detection (tripwire)
    b. Detect changes in etc
    c. Match suspicious process names
    d. Check open ports
    e. Configure sshd
    f. Blacklist processes
    g. Configure host firewall to allow NTP queries from a subnet

4. System integration - install WordPress

    a. httpd configuration
    b. DB configuration

5. "We're up a lot" - make sure services are running

    a. Make sure a single service is running
    b. Make sure services are running

6. Install PHP PECL module
7. Setting the environment for an external command
8. Using Cfengine to create and configure Amazon VMs

    a. Helper shell script to start micro instance
    b. Example of WordPress installation across two EC2 nodes (Web and DB server)
    c. System integration

9. Classes - part 3

    a. Parsing readtcp output to set a class
    b. Set a custom class based on hostname pattern
    c. Set a custom class based on hostname pattern and use a bundle of promises based on that class (separate policies based on server class)
    d. Set a custom class based on hostname pattern and import policy and run bundles based on that class (separate policy FILES based on server location)
    e. Example of using a persistent class to scan the filesystem more often in a highened security alert condition.

Part VI. Cfengine Language: Special Variables

2. What are max sizes of scalars, lists and arrays?
3. Unit tests / examples - great for learning Cfengine language
4. Orion Cloud Pack - Cfengine policy set for Amazon EC2
5. Rule of thumb: Always specify the class (classes can be implicit too!) Don't get bitten.
6. Example of an insane config (a mutually exclusive policy set)
7. Code Sharing
   a. Submitting additions to the Community Open Promise Body Library
   b. cfengine github.com site for contrib policies
   c. Index of shared Cfengine policies

```
MISC.


         g. Configuring the system resolver (DNS client)





    6. The Evolution of Cfengine: A Procession of Workable Ideas

       a. A single descriptive language that abstracts the headache
          of managing multiple Unix-like systems.

       b. Self-healing: automatic convergence to desired state;
          raising the extropy potential of the system to counteract
          the entropy which would ordinarily lead to state drift.

       c. Promise theory.  The world runs on trust.  How trust relates
          to distributed computing systems.  Predicting behavior of
          distributed systems.

       d. Knowledge Management.  KM as a challenge to scalability.
          Staff turnover can lead to loss of knowledge.  The Breadth
          versus Depth problem.  Ability to present a high level
          summary yet with ability to drill down and alter the low
          level details.


    7. Design Principles

       a. Workability/practicality is seen in all of the following
          basic design principles.

       b. Pull model, not push.  This allows for autonomy of control,
          and loose or strict federation.  (Centralization optional.)
          Human factors - handling multiple departments or business units.
          Cfengine is based on voluntary cooperation.

       c. Security.  Assumption of a hostile environment. Practical
          paranoia.  Examples: Cfengine checks if its policy files are
          world-writable; and that external commands are specified using
          absolute paths.  Defence against buffer overflows in Cfengine
          network protocol.  Cfengine's security track record (3 major
          issues in 18 years.)

       d. Resilience.  Assumption of a hostile envirnment, redux.
```

```
          Ability to withstand loss of communication with the master node.
          Local caching of policy AND binaries.

       e. Scalability.  The impressive numbers in how lightweight Cfengine
          binary is, and how many nodes can run using a single policy
          server.


    8. Cast of Characters - introducing the leads

        a. cf-agent is the piece that actually makes changes to your
           system

        b. cf-serverd is a network file server, and can be "poked"
           to run cf-agent.  cf-serverd can be used to distribute policies


    8. Some possible Cfengine architectures.

        a. The simplest - running Cfengine on a single system.

        b. A multi-node environment: one master, multiple end nodes.

        c. Cfengine at scale: a high-availability architecture
           with policy definition decoupled from policy distribution.
           [with illustration].

Misc:
              Blacklisting a process for graceful shutdown, improved:
              using regex for tighter match (so you don't kill
              "vi cupsd.conf" if you want to blacklist "cupsd".)


0070_Basic_Examples._Hello_world_using_command_bin_echo,_With_a_really_long_path.cf

///////////////////////////////////////////////////////////////////////
```