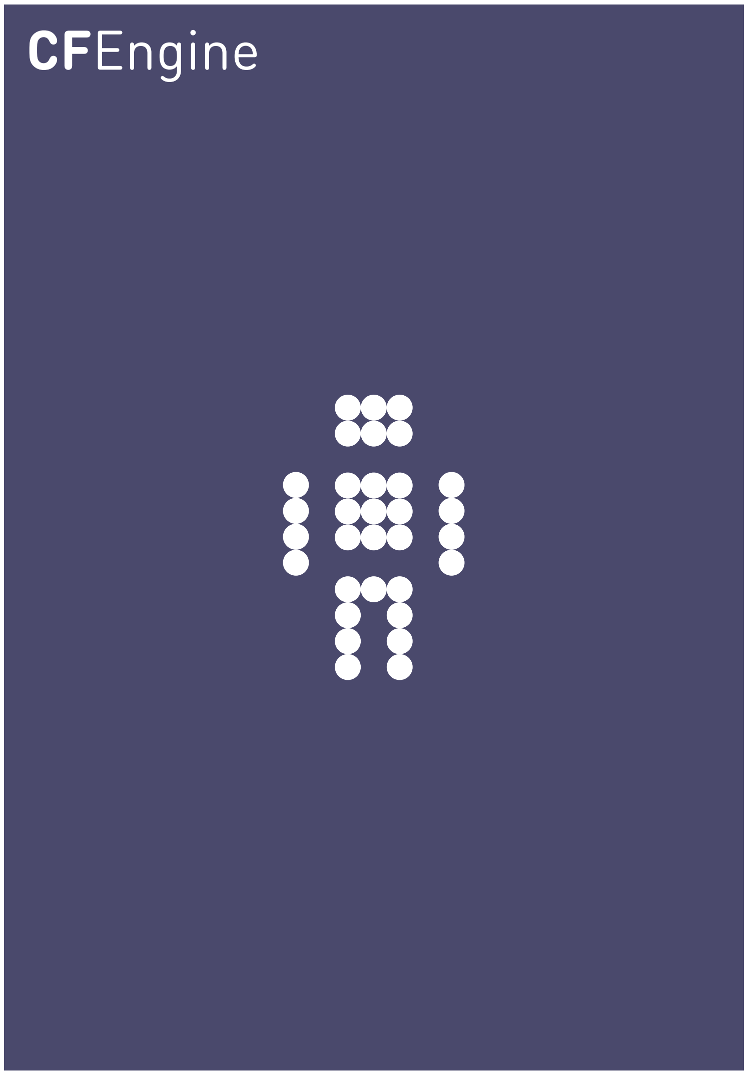

CFEngine for ‘DevOps’ and Cloud Developers

A CFEngine Special Topics Handbook

CFEngine AS

Today's Cloud model is about managing re-usable infrastructure resources; DevOps extends this to manage and customize application resources. CFEngine handles deployment, customization and repair at all levels from the operating platform to the business applications.

CFEngine has the sophistication to enable precise integration of software systems, and the responsiveness to determine the state of systems within five minutes in massive cloud-scale deployments. It runs on everything from handheld smartphones to mainframes.

Table of Contents

What is DevOps?

DevOps is a term coined by Patrick Debois in 2009, from an amalgamation of Development and Operations. It expresses a change in the way companies are thinking about IT – a change from segregated IT infrastructure to highly integrated platforms. Leading the way is a group of highly innovative Web-based companies whose businesses depend on very specific arrangements of infrastructure. It is about giving software developers more influence over the IT infrastructure their applications run on.

Why is DevOps happening now?

The proliferation of Free and Open Source software has put powerful software components in the hands of a broader range of developers than ever before – and businesses everywhere are exploiting this software by adapting it and combining it is a wealth of mutations. This blurs the line between what used to be development and what used to be the system administrator's domain (operations). We have entered an age analogous to that of hobby electronics for IT systems, where we can order off-the-shelf components and build cool new applications from them anywhere.

After 20 years of scepticism, business and Free Open Source software have made friends and are working together creatively for the benefit of willing consumers. With this basic premise of agility, companies working in this area naturally embrace a rapid innovation cycle, meaning a fast release cycle too. Traditional IT management methods can be perceived as too slow in such an environment. An important part of DevOps is that it naturally encompasses the idea of business integration – or IT for a purpose.

Should Web and IT management be closely related?

Web frameworks have seen the rise of languages like PHP, Java, Python and Ruby, all of which offer frameworks for fast deployment. Languages that work well for application development are not well suited to managing infrastructure however: they focus on details that one would like to escape from. The fact that programmers already know the languages does not change this.

An important principle for robustness and stability of systems is weak coupling between components. This brings flexibility rather than brittle fragility. Giving programmers direct control over infrastructure from their applications risks insufficient separation in which infrastructure management becomes a second-class citizen run by amateurs who just want to get code out there and don't properly understand the implications. The System Administrator role exists for a reason.

Should we use the web and HTTP for everything just because we know it? We suggest not. HTTP is an inefficient protocol for operations. It was designed for 1:1 communication with centralized certificate verification, not for decentralized 1000000:1 communication, as testified by the extensive need for load balancers in web farms.

At CFEngine, we believe in lightweight management – made as simple as possible, but no simpler.

How do we make controlled change faster?

It is important to be able to make changes quickly. Automation can implement change quickly if humans can get their acts together. Human IT processes and best practices (e.g. ITIL, COBIT, etc) tend to over bureaucratize change, leading to unnecessary overhead which frustrates agile companies.

To be confident and efficient ('less haste more speed'), there needs to be a model for the system that everyone agrees on. Models compress information and cache understanding, meaning we have less to talk about[1]. Finally, models allow us to make predictions, so they aid understanding and help us to avoid mistake.

CFEngine's promise model offers a flexible approach to weakly-coupled autonomous resource configuration. It simultaneously allows efficient convergent and repeatable implementation, and a simple definition of *compliance* with requirements[2]. All web-based companies using credit cards will know about the need for PCI-DSS compliance, for instance. And US-traded companies will know about Sarbanes-Oxley (SOX).

What role does CFEngine play in DevOps?

The challenges for IT management today are about increasing complexity (driven by the circuitry of online applications) and increasing scale.

CFEngine is not a programming language, but a documentation language for system state that has the pleasant side effect of enforcing that state on a continuous basis. It gets away from the idea of 'build automation' to complete lifecycle management. It's continuity is a natural partner for a rapid development environment, as mistakes can be quickly fixed on the fly with minimal impact on the system.

CFEngine's wins are that it is massively scalable, very low impact and rich in functionality. It will not break at a few hundred machines or choke off network communications with overhead. It will fix practically any well-defined problem within 5 minutes, bringing dependability and agility.

Knowledge, business integration - metrics

The advantage CFEngine brings is that users can have clear expectations about their systems at all times. Today's programmers are more sophisticated than script monkeys.

Getting used to declarative expression

CFEngine uses a pragmatic mixture of the declarative (functional) and imperative to represent configurations. Programmers are taught mainly imperative programming today, so a declarative approach could seem like a barrier to adoption. The principles are very simple however.

[1] Consider, for example, US versus Norwegian legal systems. In Norway more details are codified into federal law. This means that there is less to talk about in court and legal proceedings are much more quickly resolved as there is less need to reinvent interpretations on the fly.

[2] For an explanation of convergence, see the Special Topics Guide on Change Management and Incident Repair.

In spite of the focus on readability for documenting *intent*, all the familiar structures of imperative programming are, in fact, available in CFEngine, just thinly disguised for clarity.

The main goals of CFEngine are *convergence to a desired state*, *repeatability* and *clear intentions*.

Expressing actions or tasks in CFEngine

Most of the actionable items have builtin operational support, which is designed to be convergent and safely repeatable. To keep declarations clear, CFEngine organizes similar operations into chapters in a simple separation of concerns.

```
files:
 "affected object" ...details....

processes:
 "affected object" ...details....
```

In general, many such promises and types are collected into bundles, so that the form is

```
bundle agent SomeUserDefinedName
{
type_of_promise:

  "affected object/promiser"

     body of the promise/details

 ...
}
```

Expressing conditionals in CFEngine

CFEngine uses the idea of contexts (also called classes or class-contexts[3]) to address declarations to certain environments. The contexts or classes are written as a prefix, a bit like a target in a Makefile. They represent known properties of the environment.

[3] The term classes was originally used but has since been overloaded with connotations from Object Orientation, etc, making the term confusing.

```
bundle agent SomeUserDefinedName
{
type_of_promise:

  property::

    make one promise...

  !property::

    make a different promise...
}
```

This is the mechanism by which all decisions are made in CFEngine. Class contexts are evaluated by `cf-agent` and are cached so that they can be used at any time.

How do we know if the property has been evaluated or not? CFEngine evaluates certain hard-classes by default. In addition, you can probe as many more as you like, as separate promises.

```
classes:

  "cached_result" expression => fileexists("/some/file");
  "bigger"        and => { isgreaterthan("1","0"), "cached_result" };
```

This is different from a programming language where you generally make these tests in-line when you need them. In CFEngine the chance that you need the same test multiple times is greater, so the determination is separated entirely from the usage.

To go from *if-then-else* thinking to using classes, you just need to thihnk about classes as booleans:

```
bundle agent Name
{
classes:

   "cached_result" expression => fileexists("/some/file");
   "bigger"        and => { isgreaterthan("1","0"), "cached_result" };

reports:

  bigger::
    "Bigger is true....";

  cached_result&!bigger::
    "Mathematics seems to be awry...";

  # may also be written cached_result.!bigger::

}
```

These results can then be extended and reused efficiently. The class definitions can be hidden away and suitably meaningful class names replace a lot of redundant syntax.

All the information about class contexts is evaluated at the end-host, in a decentralized manner avoiding clogging of network communications that befuddles many centralized approaches. This keeps CFEngine execution very fast and with a low overhead.

Expressing loops in CFEngine

Lists and loops go hand in hand, and they are a very effective way of reducing syntax and simplifying the expression of intent. Saying ‘do this to all the following’ is generally easier to comprehend than ‘do this to the first, do this to the next,...’ and so on, because our brains are wired to see patterns.

Thus, loops are as useful for configuration as for programming. We only want to simlify the syntax once again to hide redundant words like ‘foreach’. To do this, CFEngine makes loops implicit. If you use a scalar variable reference ‘`$(mylist)`’ to a list variable ‘`@(mylist)`’, CFEngine assumes you want to iterate over each case.

```
vars:
   "my_list" slist => { "one", "two", "three" };

files:
   "/tmp/file_$(my_list)"
             create => "true";
```

The above evaluates to three promises:

```
     files:

        "/tmp/file_one"
                  create => "true";

        "/tmp/file_two"
                  create => "true";

        "/tmp/file_three"
                  create => "true";
```

Similarly the following

```
bundle agent x
{
vars:
   "hi"   string => "Hello";
   "list1" slist => { "a", "b", "c" };
   "list2" slist => { "1", "2", "3", "4" };
   "list3" slist => { "x", "y", "z" };

reports:
   !silly_non_existent_context::
     "$(hi) $(list1) $(list2) $(list3)";
}
```

Results in:

```
     R: Hello a 1 x
     R: Hello b 1 x
     R: Hello c 1 x
     R: Hello a 2 x
     R: Hello b 2 x
     R: Hello c 2 x
     R: Hello a 3 x
     R: Hello b 3 x
     R: Hello c 3 x
     R: Hello a 4 x
     R: Hello b 4 x
     R: Hello c 4 x
     R: Hello a 1 y
     R: Hello b 1 y
     R: Hello c 1 y
     R: Hello a 2 y
     R: Hello b 2 y
     R: Hello c 2 y
     R: Hello a 3 y
     R: Hello b 3 y
     R: Hello c 3 y
     R: Hello a 4 y
     R: Hello b 4 y
     R: Hello c 4 y
     R: Hello a 1 z
     R: Hello b 1 z
     R: Hello c 1 z
     R: Hello a 2 z
     R: Hello b 2 z
     R: Hello c 2 z
     R: Hello a 3 z
     R: Hello b 3 z
     R: Hello c 3 z
     R: Hello a 4 z
     R: Hello b 4 z
     R: Hello c 4 z
```

Expressing subroutines in CFEngine

Subroutines are used for both expressing and reusing parameterizable chunks of code, and for naming chunks for better management of intention. In CFEngine you define these as `methods`. A method is simply a bundle of promises, possibly with parameters. To call a method, you make a method-use-bundle promise. In this example, we call a bundle called `subtest` which accepts a parameter from its calling bundle.

```
body common control
{
# Master execution list
bundlesequence  => { "testbundle"  };
}
```

```
###########################################

bundle agent testbundle
{
vars:
 "userlist" slist => { "one", "two", "three" };

methods:
 "any" usebundle => subtest("$(userlist)");
}

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

bundle agent subtest(user)
{
commands:
  "/bin/echo Fix $(user)";
}
```

The use of methods brings multi-dimensional patterns to convergent configuration management.

Using CFEngine to integrate software components

Integration of software components may be addressed with a variety of approaches and techniques:

- Standard template methods from the COPBL community library ('out of the box' solutions).
- Customized, personalized configurations.
- Package management for software dependencies.
- File management - copying, editing, permissions, etc.
- Process management - starting, stopping, restarting.
- Security.
- Monitoring performance and change.

Needless to say, all of these are easily achievable with 5 minute repair accuracy using our CFEngine framework.

Cloud computing is a rehearsal

We have barely made a dent in CFEngine in this Short Topics Guide. Let us end by noting briefly that DevOps and Cloud Computing are merely rehearsals for what is to come next: molecular computing in which we synthesize complex clusters of components based on higher level rule based schemas.

In this future version of IT, knowledge management will be the key challenge for understanding how to build systems. We fully expect the APIs of the future virtualized infrastructure to be promise oriented, and for CFEngine to remain a viable approach to configuration after other frameworks have become outmoded.