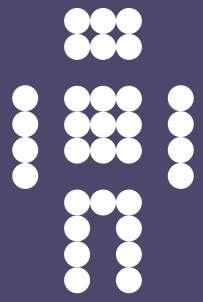
CFEngine



Application Management

A CFEngine Special Topics Handbook

CFEngine AS

CFEngine is able to install, update and uninstall services and applications across all managed nodes in a platform-independent manner.

Updating software across all nodes can be made as simple as copying a package to a software server. Thereafter, applications can be managed and customized using CFEngine.

Table of Contents

What is Application Management?

Application management concerns the deployment and updating of software, as well as customization of for actual use, in other words all the activities required to make an application ready for use. Initially, software installation packages must be deployed on host machines, however, we frequently encounter the need to update software due to security flaws, bugs or new features.

It is generally unwise to let every application update itself automatically to the newest version from the internet; we want to decide which version gets installed and also make sure that the load on the network does not impair performance during mass-updates. Equally important is making sure certain applications are not present, especially when they are known to have security issues.

Using CFEngine, you can verify that the software is in a promised state and is properly customized for use.

How can CFEngine help?

CFEngine assists with application management in a number of ways. Following the BDMA lifecycle, we note:

Build CFEngine can be used to automate the build of packaged software releases using standardized or custom package formats.

Deploy CFEngine can distribute and install packaged software on any kind of platform.

Manage CFEngine can start, stop, restart, monitor, and upgrade, and customize software applications.

Audit CFEngine can monitor and report on packages and patches installed on systems and their versions and status.

Package management

Application management is simple today on most operating systems due to the introduction of *package systems*.

All major operating systems now have some sort of package management system, e.g. RPM for Linux, and MSI for Windows. However, their capabilities and methods vary greatly. Moreover, the packages they need to install have to be made available to the hosts that need them and the package manager has to be executed at the right time and place. This is where CFEngine assists.

Some package managers support online automatic access of online repositories and can download data from the network. Others have to have packages copied to local storage first. CFEngine can work with both types of system to integrate software management.

- CFEngine communicates with the system using its own standards to utilize the approach suitable for that software system.

- Custom software repositories can be made, and CFEngine's agents can perform this distribution by collecting software packages to local storage and then installing from there.

When software packages are available on local storage, CFEngine can check whether they are already installed, and if so, which version and architecture are installed. This, in turn, can be verified against the policy for the software — should it indeed be installed, updated or removed?

Using the CFEngine standard library, agents know how to talk to the native package manager to query information and get the system into the desired state.

CFEngine can edit configuration files in real time to ensure that applications are customized to local needs at all times.

Enterprise Software Reporting

In commercial releases of CFEngine, the state of software installation is reported centrally and is easily accessible through the Knowledge Map.

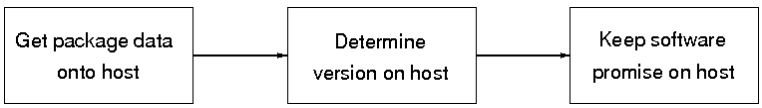
Commercial editions of CFEngine also support querying Windows machines for installed MSI packages and thus allows for easy software deployment in heterogeneous Unix and Windows environments.

Integrated software installation

CFEngine gives complete freedom to users, so there are many ways to design a system that achieves a desired software end-state. Consider the following example setup which ensures that one particular application is up to date on all hosts. The procedure below is very similar to the way that commercial CFEngine editions update.

Rather than using an OS-specific package repository, like yum, we create a universal approach using CFEngine's distribution and installation promises.

We first look at the example on an RPM system, then we show the modifications required to handle Windows instead. The examples use body parts from the standard library.



Distributing software packages to client hosts

To begin with, we promise that the relevant software packages will be locally available to the agents from software servers, i.e. we promise that a local copy of all deployed software packages will exist in the directory '`/software_repo`' on local storage. The copy will be collected and compared against a directory called '`/master_software_repo`' on host `server.example.org` in this example.

We say that this approach is 'data-driven' because, by placing software package data in the central repository, client hosts update automatically, as they promise to subscribe to the data.

```
files:

  "/software_repo"

      comment => "Copy app1 updates from software server",
    copy_from => remote_cp("/master_software_repo/app1/$(sys.flavour)",
                           "server.example.org"),
  depth_search => recurse("inf"),
       classes => if_repaired("newpkg_app1");
```

When the agent copies a relevant software package from the software server (`sys.flavour` is the local operating system), the class `newpkg_app1` will get defined. This class can act as a trigger to stop the application, update it, and start it again.

Stopping and restarting an application for update

On some operating systems, software cannot be updated while it is running. CFEngine can promise to enure that a program is stopped before update:

```
processes:

 newpkg_app1::

  "app1" signals => { "term", "kill" };
```

CFEngine *normal ordering*, ensures that `processes` promises are always run prior to `packages` promises, so the application will be stopped before updated. Next we promise the version of the software we want to install. In this case, any version greater than 1.0.0.

```
packages:

 newpkg_app1::

  "app1"

    package_policy           => "update",
    package_select           => ">=",
    package_architectures    => { "i586" },
    package_version          => "1.0.0",
    package_method           => rpm_version("/software_repo"),
    classes                  => if_else("app1_update", "app1_noupdate");
```

By promising carefully what package and version you want, using `package_policy`, `package_select`, and `package_version`, CFEngine can keep this promise by updating to the latest version of the package available in the directory repository '`/software_repo`'. If the available versions are all 'less than' than "1.0.0", an update will not take place.

The package_version specification should match the versioning format of the software, whatever it is, e.g. you would write something like "1.00.00.0" if two digits were used in the two middle version number positions.

CFEngine automatically adapts its versioning to the conventions used by individual package schemas.

To summarize, in order for CFEngine to be able to match installed packages with the ones in the directory repository, the same naming convention must be applied. That is, the package name, version and architecture must have the same format in the list of installed packages as the file names of available packages.

From the promise above, we see that CFEngine will interpret app1 as the name, 1.0.0 as the version and i586 as the architecture of the package. Using this while looking at the package_name_convention in the rpm package method, we see that CFEngine will look for packages named as app1-X.Y.Z-i586.rpm, with X, Y, Z producing the largest version available in the directory repository. If an available version is larger than the one installed, an update will take place — the update command is run.

Finally, we set classes from the software update in case we want to act differently depending on the outcome.

Replacing the policy 'update' with 'add' is all that is required to install the package (once) instead of updating. Using policy 'add' will do nothing if the package is already installed, but installs the largest version available if it is not. Use package_select => "==" to install the exact version instead of the largest.

Adapting to Windows

To adapt our example to Windows, we change the path to the local software repository from '/software_repo' to 'c:\software_repo', to support the Windows path format. Other than that, all we have to change is the package_method, yielding the following.

```
package_method           => msi_version("c:\software_repo"),
```

Refer to the msi_version body in the standard library.

Notes on Windows systems

CFEngine implements Windows packaging using the MSI subsystem, internally querying the Windows Management Interface for information. However, not all Windows systems have the reqired information.

CFEngine relies on the name (lower-cased with spaces replaced by hyphen) and version fields found inside the msi packages to look for upgrades in the package repository.

Problems can arise when the format of these fields differ from their format in the file names. For example, a package file name may be 7zip-4.65-x86_64.msi, while the product name in the msi is given as 7-Zip 4.65 (x64 edition), and the version is 4.65.00.0.

For the formats to match, we can change the product name to `7zip` and the version to `4.65` in the msi-package. Free tools such as `InstEd` can both view and change the product name and version (Tables->Property->ProductName and ProductVersion).

Customizing applications

By definition, we cannot explain how to customize software for all cases. For Unix-like systems however, software customization is usually a matter of editing a configuration text file. CFEngine can edit files, for instance, to add a configuration line to a file, you might so something like this:

```
bundle agent my_application_customize
{
files:

  "$(prefix)/config.cf"

        comment => "Set the permissions and add a line...",
      perms     => mo("0600","root"),
      edit_line => append_if_no_line("My custom setting...");
}
```

To set a number of variables inside a file, you might do something like this:

```
bundle agent my_application_customize
{
vars:

  # want to set these values by the names of their array keys

  "rhs[serverhost]" string => "123.456.789.123";
  "rhs[portnumber]" string => "1234";
  "rhs[admin]"      string => "admin@example.org";

files:

  "$(prefix)/config.cf"

       comment => "Add new variables or set existing ones",
     edit_line => set_variable_values("setvars.rhs");

}
```

You can also create file templates with customizable variables using the `expand_template` method from the standard library.

Starting and stopping software

CFEngine is promise or compliance oriented. You promise whether software will be running or not running at different times and locations by making `processes` or `services` promises.

To start a service, you might do something like this:

```
processes:

   "myprocess"  restart_class => "start_me";

commands:

  start_me::

    "/path/to/software"

         # ... many security options, etc
```

or using services

```
services:

 windows::

  "Dhcp"
    service_policy => "start",
    service_dependencies => { "Alerter", "W32Time" },
    service_method => winmethod;
```

To stop a service, you take one of these approaches:

```
processes:

   "badprocess"
      signals => { "term", "kill" };

   "snmp"
      process_stop => "/etc/init.d/snmp stop";
```

Auditing software applications

Commercial Editions of CFEngine generate reports about installed software, showing package names and versions that are installed. There is a huge variety in the functionality offered by different package systems. The most sophisticated package managers are those provided by OpenSuSE Linux and RedHat. These know the difference between installation packages and software updates and can keep track of installed software transparently. Most package systems have fewer functions.

CFEngine tries to make the best of each package system to collect information about the state of software. In commercial editions you have access to reports on the software installed on each system in the network, to the extent permitted by the software subsystems on those hosts.