cuisine Documentation

Release

2011, Sebastien Pierre

April 20, 2015

```
           /      /
 ___         ___    ___  ___
|    |   )| |___ | |   )|___)
|__  |__/ |  __/ | |  / |__

-- Chef-like functionality for Fabric
```

Fabric is an incredible tool to automate administration of remote machines. As Fabric's functions are rather low-level, you'll probably quickly see a need for more high-level functions such as add/remove users and groups, install/upgrade packages, etc.

Cuisine is a small set of functions that sit on top of Fabric, to abstract common administration operations such as file/dir operations, user/group creation, package install/upgrade, making it easier to write portable administration and deployment scripts.

Cuisine's features are:

- Small, easy to read, a single file API: <object>_<operation>() e.g. dir_exists(location) tells if there is a remote directory at the given location.
- Covers file/dir operations, user/group operations, package operations
- Text processing and template functions
- All functions are lazy: they will actually only do things when the change is required.

CHAPTER 1

Installation

Cuisine is on PyPI so you can either use `easy_install -U cuisine` or `pip install cuisine` to install it. Otherwise, you can download the source from GitHub and run `python setup.py install`.

CHAPTER 2

How to get started

Open up a python shell and type:

```
import cuisine
```

Cuisine is designed to be a flat-file module, where all functions are prefixed by the type of functionality they offer (e.g., *file* for file operations, *user* for user operations). The available groups are:

text_* Text-processing functions

file_* File operations

dir_* Directory operations

package_* Package management operations

command_* Shell commands availability

user_* User creation commands

group* Group creation commands

mode_* Configures cuisine's behaviour within the current session.

select_* Selects a specific option, such as package back-end (*apt*, *yum*, *zypper*, or *pacman*)

If you're using an interactive Python shell such as IPython you can easily browse the available functions by using tab-completion.

```
In [2]: cuisine.file_
cuisine.file_append       cuisine.file_is_file      cuisine.file_unlink
cuisine.file_attribs      cuisine.file_is_link      cuisine.file_update
cuisine.file_attribs_get  cuisine.file_link         cuisine.file_upload
cuisine.file_ensure       cuisine.file_local_read   cuisine.file_write
cuisine.file_exists       cuisine.file_read
cuisine.file_is_dir       cuisine.file_sha256
```

As the functions are prefixed by they type of functionality, it is very easy to get started using an interactive shell.

If you would like to use cuisine without using a *fabfile*, you can call the *mode_local()* function.

```
import cuisine
cuisine.mode_local()
print cuisine.run("echo Hello")
```

alternatively, you can also directly connect to a server

```
import cuisine
cuisine.connect("my.server.com")
print cuisine.run("echo Hello")
```

If you want to use cuisine within a *fabfile*, simply create a *fabfile* with the following content:

```
from cuisine import *

def setup():
    group_ensure("remote_admin")
    user_ensure("admin")
    group_user_ensure("remote_admin", "admin")
```

CHAPTER 3

Troubleshooting

If you are encoutering problems, please check the following:

- The user@host is running an SH-compatible shell (sh, dash, bash, zsh should work)
- The system has *openssl base64*, *md5sum* and *sha1sum* commands in addition to the basic UNIX ones.

If you still have a problem, simply file a bug report here https://github.com/sebastien/cuisine/issues

Right now, cuisine is tested on Ubuntu. Some contributors use it on RHEL and CentOS. If you use on a different system, let us know if it works!

Contributing specific implementations

Cuisine was originally developed as a Debian/Ubuntu-centric tool, but can easily be adapted to other distributions or Unix flavor, the only caveat being that the shell is expected to be bash-compatible.

If you want to implement a specific variant of some functions for a specific platform, you should do the following:

1. Open the *cuisine.py* source and look for the definition of the function that you would like to specialize.
2. If the function is decorated by '@dispatch', it means it already supports specific back-ends (see *package_** functions), and you can proceed to the next step. Otherwise, you can either file a ticket on Github or read the source and mimic what we've done for *package_**
3. Create a specific version of the decorated function by creating a new function with the same name, suffixed by your specific backend name. For instance, if you'd like to create a *yum* backend to *package_ensure*, you would need to create a function *package_ensure_yum* with the same arguments as *package_ensure*
4. Once you've created your specific functions, make sure that you have a *select_** matching your function group. For the *package_** functions, this would be *select_package*.
5. Look for the *supported* variable in the *select_** and add your backend suffix to it (in our example, this would be *yum*)

To use a specific backend implementation of a set of features, use the *select_** functions.

```
# To use the 'apt' backend
cuisine.select_package("apt")
# To see the available backends
print cuisine.select_package()
```

CHAPTER 5

Modules

Cuisine-PostgreSQL http://pypi.python.org/pypi/cuisine-postgresql/

CHAPTER 6

More?

If you want more information, you can:

- Read the presentation on Cuisine
- Read Cuisine: the Lightweight Chef/Puppet Alternative