Etcaetera Documentation

Release

Oleiade

April 08, 2014

CHAPTER 1

What?

Etcaetera helps you loading your application configuration from multiple sources in a simple way. It exposes a single **Config** object which you add prioritized sources adapters to (*env*, *files*, *cmdline*, *modules*...).

Once you call `load` method over it: your settings are loaded from your adapters in order, all your configuration is stored in the **Config** object.

You're **done**.

CHAPTER 2

Why?

Managing a large application configuration sources can be a pain in the neck. Command line, files, system environment, modules, a lot of mixed sources can provide you with the settings you seek.

They are all accessed in different ways, and establishing a merging strategy of these differents sources can sometimes look like impossible. Etcaetera provides you a simple and unified way to handle all the complexity in a single place.

2.1 How to

2.1.1 Installation

With pip

```
$ pip install etcaetera
```

With setuptools

```
$ git clone git@github.com:oleiade/etcaetera
$ cd etcaetera
$ python setup.py install
```

2.1.2 Usage

Dive

A real world example is worth a thousand words

```
>>> from etcaetera.config import Config
>>> from etcaetera.adapters import Defaults, Module, Overrides, Env, File

# Let's create a new configuration object
>>> config = Config()

# And create a bunch of adapters
>>> env_adapter = Env("MY_FIRST_SETTING", "MY_SECOND_SETTING")
>>> python_file_adapter = File('/etc/my/python/settings.py')
>>> json_file_adapter = File('/etc/my_json_settings.json')
>>> module_adapter = Module(os)
>>> overrides = Overrides({"MY_FIRST_SETTING": "my forced value"})

# Let's register them
>>> config.register(env_adapter, python_file_adapter, json_file_adapter, module_adapter, overrides)

# Load configuration
>>> config.load()


# And that's it
>>> print config
{
    "MY_FIRST_SETTING": "my forced value",
    "MY_SECOND_SETTING": "my second value",
    "FIRST_YAML_SETTING": "first yaml setting value found in yaml settings",
    "FIRST_JSON_SETTING": "first json setting value found in json settings",
    ...
}
```

Config object

The config object is the central place for your whole application settings. It loads your adapters in the order you've registered them, and updates itself using it's data. Furthermore you can attach sub config objects to it, in order to keep a clean configuration hierarchy.

Please note that **Defaults** adapter will always be loaded first, and **Overrides** will always be loaded last.

```
>>> from etcaetera.config import Config
>>> from etcaetera.adapters import Defaults, Module, Overrides, Env, File

# Let's create a new configuration object
>>> config = Config()

# And create a bunch of adapters
>>> env_adapter = Env("MY_FIRST_SETTING", "MY_SECOND_SETTING")
>>> python_file_adapter = File('/etc/my/python/settings.py')
>>> json_file_adapter = File('/etc/my_json_settings.json')
>>> module_adapter = Module(os)
>>> overrides = Overrides({"MY_FIRST_SETTING": "my forced value"})

# Let's register them
>>> config.register(env_adapter, python_file_adapter, json_file_adapter, module_adapter, overrides)

# Load configuration
>>> config.load()


# And that's it
>>> print config
{
    "MY_FIRST_SETTING": "my forced value",
    "MY_SECOND_SETTING": "my second value",
    "FIRST_YAML_SETTING": "first yaml setting value found in yaml settings",
    "FIRST_JSON_SETTING": "first json setting value found in json settings",
    ...
}
```

```
# If you need a certain hierarchy for your configuration
# Config objects supports sub configs. Here's an example of
# how to add an "aws" subconfig
>>> aws_config = Config()  # Create a config obj
>>> aws_env = Env("AWS_ACCESS_KEY_ID", "AWS_SECRET_ACCESS_KEY")
>>> aws_config.register(aws_env)  # Register an env adapter on to it
>>> config.add_subconfig('aws', aws_config)
>>> config.aws
{
    "AWS_ACCESS_KEY_ID": "128u09ijod019jhd182o1290d81",
    "AWS_SECRET_ACCESS_KEY": "qoiejdn0182hern1d098uj12podij1029udaiwjJBIU09u0oimJHKI"
}

# Config objects are also able to automatically format the keys incoming
# from adapters. For example if you'd want all your keys to be lowercased
# just pass it the appropriate formatter (from etcaetera.formatters import *)
>>> from etcaetera.formatters import lowercased
>>> formatted_config = Config(formatter=lowercased)
>>> env_adapter = Env(**{"USER": "SUPER_DUPER_USER"})
>>> formatted_config.register(env_adapter)
>>> formatted_config.load()
>>> print formatted_config
{
    "super_duper_user": "oleiade",
}
```

Adapters

Adapters are the interfaces with configuration sources. They load settings from their custom source type, and they expose them as a normalized dict to *Config* objects.

Right now, etcaetera provides the following adapters:

- *Defaults*: sets some default settings
- *Overrides*: overrides the config settings values
- *Env*: extracts configuration values from system environment
- *File*: extracts configuration values from a file. Accepted format are: json, yaml, python module file (see *File adapter* section for more details)
- *Module*: extracts configuration values from a python module. Like in django, only uppercased variables will be matched

In a close future, etcaetera may provide adapters for:

- *Argv* argparse format support: would load settings from an argparser parser attributes
- *File* ini format support: would load settings from an ini file

Cool features you should know about:

- You can provide a *formatter* to your adapters so the imported keys will be automatically modified. Example `Env("USER", etcaetera.formatters.lowercased)` will import the `$USER` environment variable as `user` when `.load()` is called.

Defaults adapter

Defaults adapter provides your configuration object with default values. It will always be evaluated first when `Config.load` method is called. You can whether provide defaults values to *Config* as a *Defaults* object or as a dictionary.

```
>>> from etcaetera.adapter import Defaults

# Defaults adapter provides default configuration settings
>>> defaults = Defaults({"ABC": "123"})
>>> config = Config(defaults)

>>> print config
{
    "ABC": "123"
}
```

Overrides adapter

The Overrides adapter overrides *Config* object values with it's own values. It will always be evaluated last when the `Config.load` method is called.

```
>>> from etcaetera.adapter import Overrides

# The Overrides adapter helps you set overriding configuration settings.
# When registered over a Config objects, it will always be evaluated last.
# Use it if you wish to force some config values.
>>> overrides_adapter = Overrides({"USER": "overrided value"})
>>> config = Config({
    "USER": "default_value",
    "FIRST_SETTING": "first setting value"
})

>>> config.register(overrides_default)
>>> config.load()

>>> print config
{
    "USER": "overrided user",
    "FIRST_SETTING": "first setting value"
}
```

Env adapter

Env adapter loads configuration variables values from system environment. You can whether provide it a list of keys to be fetched from environment. Or you can pass it a *environment variables name to adapter destination name* `**mappings` dict. Moreover, as adapters support nested keys through the `.` separator you can map any env var to a nested adapter destination.

```
>>> from etcaetera.adapter import Env

# You can provide keys to be fetched by the adapter at construction
# as keys
>>> env = Env("USER", "PATH")
>>> env.load()
>>> print env.data
{
    "USER": "user extracted from environment",
    "PATH": "path extracted from environment",
    "PWD": "pwd extracted from environment"
}

# alternatively pass it as env var names to adapter var
# names dict
>>> os.environ["SOURCE"], os.environ["OTHER_SOURCE"]
("my first value", "my second value")
>>> env = Env({"SOURCE": "DEST", "OTHER_SOURCE": "TEST"})
>>> env.load()
>>> print env.data
{
    "DEST": "my first value",
    "TEST": "my second value"
}

# Adapters support nested destination too
>>> env = Env({"MY.USER": "USER"})
>>> env.load()
>>> print env.data
{
    "MY": {
        "USER": "oleiade",
    }
}
```

File adapter

The File adapter will load the configuration settings from a file. Supported formats are json, yaml and python module files. Every key-value pairs stored in the pointed file will be loaded in the *Config* object it is registered to.

Python module files The Python module files should be in the same format as the Django settings files. Only uppercased variables will be loaded. Any python data structures can be used.

Here's an example

Given the following settings.py file

```
$ cat /my/settings.py
FIRST_SETTING = 123
SECOND_SETTING = "this is the second value"
THIRD_SETTING = {"easy as": "do re mi"}
ignored_value = "this will be ignore"
```

File adapter output will look like this:

```
>>> from etcaetera.adapter import File

>>> file = File('/my/settings.py')
>>> file.load()

>>> print file.data
{
    "FIRST_SETTING": 123,
    "SECOND_SETTING": "this is the second value",
    "THIRD_SETTING": {"easy as": "do re mi"}
}
```

Serialized files (aka json and yaml) *Given the following json file content*:

```
$ cat /my/json/file.json
{
    "FIRST_SETTING": "first json file extracted setting",
    "SECOND_SETTING": "second json file extracted setting"
}
```

The File adapter output will look like this:

```
>>> from etcaetera.adapter import File

# The File adapter awaits on a file path at construction.
# All you have to do then, is to let the magic happen
>>> file = File('/my/json/file.json')
>>> file.load()

>>> print file.data
{
    "FIRST_SETTING": "first json file extracted setting",
    "SECOND_SETTING": "second json file extracted setting"
}
```

Module adapter

The Module adapter will load settings from a python module. It emulates the django settings module loading behavior, so that every uppercased locals of the module is matched.

Given a mymodule.settings module looking this:

```
MY_FIRST_SETTING = 123
MY_SECOND_SETTING = "abc"
```

Loaded module data will look like this:

```
>>> from etcaetera.adapter import Module

# It will extract all of the module's uppercased local variables
>>> module = Module(mymodule.settings)
>>> module.load()

>>> print module.data
{
    MY_FIRST_SETTING = 123
    MY_SECOND_SETTING = "abc"
}
```

2.2 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.2.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/oleiade/etcaetera/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Etcaetera could always use more documentation, whether as part of the official Etcaetera docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/oleiade/etcaetera/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.2.2 Get Started!

Ready to contribute? Here's how to set up *etcaetera* for local development.

1. Fork the *etcaetera* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/etcaetera.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv etcaetera
$ cd etcaetera/
$ python setup.py develop
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 etcaetera tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request to the *develop* branch through the GitHub website.

2.2.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/oleiade/etcaetera/pull_requests and make sure that the tests pass for all supported Python versions.

2.3 History

2.3.1 0.1.0 (2014-01-11)

- First release on PyPI.