config$_r esolver$

Release 4.2.2

Mar 16, 2018

Contents

CHAPTER 1

Table of Contents

1.1 Introduction

1.1.1 config_resolver

Full Documentation https://config-resolver.readthedocs.org/en/latest/

Repository https://github.com/exhuma/config_resolver

PyPI https://pypi.python.org/pypi/config_resolver

Rationale

Many of the larger frameworks (not only web frameworks) offer their own configuration management. But it looks different everywhere. Both in code and in usage later on. Additionally, the operating system usually has some default, predictable place to look for configuration values. On Linux, this is `/etc` and the XDG Base Dir Spec (This instance is based on 0.8 of this spec).

The code for finding these config files is always the same. But finding config files can be more interesting than that:

- If config files contain passwords, the application should issue appropriate warnings if it encounters an insecure file and refuse to load it.
- The expected structure in the config file can be versioned (think: schema). If an application is upgraded and expects new values to exist in an old version file, it should notify the user.
- It should be possible to override the configuration per installed instance, even per execution.

`config_resolver` tackles all these challenges in a simple-to-use drop-in module. The module uses no additional external modules (no additional dependencies, pure Python) so it can be used in any application without adding unnecessary bloat.

Description / Usage

The module provides one function to retrieve a config instance:

- `get_config()`

and one function to create a config from a text-string:

- `from_string()`

A simple usage looks like this:

```
from config_resolver imoprt get_config
result = get_config('acmecorp', 'bird_feeder')
cfg = result.config
```

This will look for config files in (in that order):

- `/etc/acmecorp/bird_feeder/app.ini`
- `/etc/xdg/acmecorp/bird_feeder/app.ini`
- `~/.config/acmecorp/bird_feeder/app.ini`
- `./.acmecorp/bird_feeder/app.ini`

If all files exist, one which is loaded later, will override the values of an earlier file. No values will be removed, this means you can put system-wide defaults in `/etc` and specialise/override from there.

Note: The above is true for the file handlers included with `config_resolver`. Since version 5.0 it is possible to provide custom file-handlers, which may behave differently. If using a custom file-handler make sure to understand how it behaves! See *Custom Handlers*.

The Freedesktop XDG standard

freedesktop.org standardises the location of configuration files in the XDG specification Since version 4.1.0, `config_resolver` reads these paths as well, and honors the defined environment variables. To ensure backwards compatibility, those paths have only been added to the resolution order. They have a higher precedence than the old locations though. So the following applies:

XDG item	overrides
`/etc/xdg/<group>/<app>`	`/etc/<group>/<app>`
`~/.config/<group>/</app>`	`~/.<group>/<app>`
`$XDG_DATA_HOME`	`$GROUP_APP_PATH`
`$XDG_CONFIG_DIRS`	`$GROUP_APP_PATH`

By default, files are parsed using the default Python `configparser.ConfigParser` (i.e. `ini` files). Custom file "handlers" may read other formats. See *Custom Handlers*.

Advanced Usage

The way config_resolver finds files can be controlled by an optional `lookup_options` argument to `get_config()`. This is a dictionary controlling how the files are searched and which files are valid. The default options are:

```
default_options = {
    'search_path': [],  # <- empty list here triggers the default search path
    'filename': 'app.ini',  # <- this depends on the file-handler
    'require_load': False,
    'version': None,
    'secure': False,
}
```

All values in the dictionary are optional. Not all values have to be supplied. Missing values will use the default value shown above.

Versioning

It is pretty much always useful to keep track of the expected "schema" of a config file. If in a later version of your application, you decide to change a configuration value's name, remove a variable, or require a new one the end-user needs to be notified.

For this use-case, you can use the lookup option `version` to allow only files of the proper version to be loaded. If the version differs in a detected file, a log message will be emitted:

```
result = get_config('group', 'app', {'version': '2.1'})
```

Config file example:

```
[meta]
version=2.1

[database]
dsn=foobar
```

If you don't specify a version number in the construcor versioning will trigger automatically on the first file encountered which has a version number. The reason this triggers is to prevent accidentally loading files which incompatible version.

Only "major" and "minor" numbers are supported. If the application encounters a file with a different "major" value, it will emit a log message with severity `ERROR` and the file will be skipped. Differences in minor numbers are only logged with a "warning" level but the file will be loaded.

Rule of thumb: If your application accepts a new config value, but can function just fine with previous and default values, increment the minor number. If on the other hand, something has changed, and the user needs to change the config file, increment the major number.

Requiring files (bail out if no config is found)

Since version 3.3.0, you have a bit more control about how files are loaded. The `get_config()` function takes the lookup_options value `require_load`. If this is set to `True`, an `OSError` is raised if no config file was loaded. Alternatively, and, purely a matter of taste, you can leave this on it's default `False` value and inspect the `loaded_files` attribute on the `meta` attribute of the returned result. If it's empty, nothing has been loaded.

Overriding internal defaults

Both the search path and the basename of the file (`app.ini`) can be overridden by the application developer via the API and by the end-user via environment variables.

By the application developer

Apart from the "group name" and "application name", the `get_config()` function accepts `search_path` and `filename` as values in `lookup_options`. `search_path` controls to what folders are searched for config files, `filename` controls the basename of the config file. `filename` is especially useful if you want to separate different concepts into different files:

```
app_cfg = get_config('acmecorp', 'bird_feeder').config
db_cfg = get_config('acmecorp', 'bird_feeder', {'filename': 'db.ini'})
```

By the end-user

The end-user has access to two environment variables:

- `<GROUP_NAME>_<APP_NAME>_PATH` overrides the default search path.
- `XDG_CONFIG_HOME` overrides the path considered as "home" locations for config files (default = `~/.config`)
- `XDG_CONFIG_DIRS` overrides additional path elements as recommended by the freedesktop.org XDG basedir spec. Paths are separated by `:` and are sorted with descending precedence (leftmost is the most important one).
- `<GROUP_NAME>_<APP_NAME>_FILENAME` overrides the default basename of the config file (default = `app.ini`).

Logging

All operations are logged using the default `logging` package with a logger with the name `config_resolver`. All operational logs (opening/reading file) are logged with the `INFO` level. The log messages include the absolute names of the loaded files. If a file is not loadable, a `WARNING` message is emitted. It also contains a couple of `DEBUG` messages. If you want to see those messages on-screen you could do the following:

```
import logging
from config_resolver import Config
logging.basicConfig(level=logging.DEBUG)
conf = get_config('mycompany', 'myapplication').config
```

If you want to use the `INFO` level in your application, but silence only the config_resolver logs, add the following to your code:

```
logging.getLogger('config_resolver').setLevel(logging.WARNING)
```

As of version 4.2.0, all log messages are prefixed with the group and application name. This helps identifying log messages if multiple packages in your application use `config_resolver`. The prefix filter can be accessed via the "meta" member `prefix_filter` if you want to change or remove it:

```
from config_resolver import Config
conf = get_config('mycompany', 'myapplication')
print(conf.meta.prefix_filter)
```

More detailed information about logging is out of the scope of this document. Consider reading the logging tutorial of the official Python docs.

Environment Variables

The resolver can also be manipulated using environment variables to allow different values for different running instances. The variable names are all upper-case and are prefixed with both group- and application-name.

`<group_name>_<app_name>_PATH` The search path for config files. You can specify multiple paths by separating it by the system's path separator default (`:` on Linux).

If the path is prefixed with +, then the path elements are *appended* to the default search path.

`<group_name>_<app_name>_FILENAME` The file name of the config file. Note that this should *not* be given with leading path elements. It should simply be a file basename (f.ex.: `my_config.ini`)

`XDG_CONFIG_HOME` and `XDG_CONFIG_DIRS` See the XDG specification

Debugging

Calling `get_config()` will not raise an error (except if explicitly asked to do so). Instead it will always return a valid, (but possibly empty) instance. So errors can be hard to see sometimes.

The idea behind this, is to encourage you to have sensible default values, so that the application can run, even without configuration.

Your first stop should be to configure logging and look at the emitted messages.

In order to determine whether any config file was loaded, you can look into the `loaded_files` "meta" variable. It contains a list of all the loaded files, in the order of loading. If that list is empty, no config has been found. Also remember that the order is important. Later elements will override values from earlier elements (depending of the used `handler`).

Additionally, another "meta" variable named `active_path` represents the search path after processing of environment variables and runtime parameters. This may also be useful to display information to the end-user.

1.1.2 Examples

A simple config instance (with logging):

```
import logging
from config_resolver import get_config

logging.basicConfig(level=logging.DEBUG)
cfg = get_config("acmecorp", "bird_feeder").config
print(cfg.get('section', 'var'))
```

An instance which will not load unsecured files:

```
import logging
from config_resolver import get_config

logging.basicConfig(level=logging.DEBUG)
cfg = get_config("acmecorp", "bird_feeder", {"secure": True}).config
print(cfg.get('section', 'var'))
```

Loading a versioned config file:

```
import logging
from config_resolver import get_config

```

```
logging.basicConfig(level=logging.DEBUG)
cfg = get_config("acmecorp", "bird_feeder", {"version": "1.0"}).config
print(cfg.get('section', 'var'))
```

Inspect the “meta” variables:

```
from config_resolver import get_config

cfg = get_config("acmecorp", "bird_feeder")
print(cfg.meta)
```

1.2 Changelog

1.2.1 Release 5.0.0

Warning: Major API changes! Read the full documentation before upgrading!

- Python 2 support is now dropped!
- Add the possibility to supply a custom file “handler” (f.ex. YAML or other custom parsers).
- Add `config_resolver.handler.json` as optional file-handler.
- Refactored from a simple module to a full-fledged Python package
- Retrieving a config instance no longer returns a subclass of the `configparser.ConfigParser` class. Instead, it will return whatever the supplied handler creates.
- External API changed to a functional API. You no longer call the `Config` constructor, but instead use the `get_config()` function.
- Retrieval meta-data is returned along-side the retrieved config. This separation allows a custom handler to return any type without impacting the internal logic of `config_resolver`.
- Dropped the deprectaed lookup in `~/.group-name/app-name` in favor of the XDG standar `~/.config/group-name/app-name`.

Upgrading from 4.x

- Replace `Config` with `get_config`
- The result from the call to `get_config` now returns two objects: The config instance and additional metadata.
- The following attributes moved to the meta-data object:
  - `active_path`
  - `prefix_filter`
  - `loaded_files`
- Return types for INI files is now a standard library instance of `configparser.ConfigParser`. This means that the `default` keyword argument to `get` has been replaced with `fallback`.

1.2.2 Release 4.2.0

Features added

- GROUP and APP names are now included in the log messages.

1.2.3 Release 4.1.0

Features added

- XDG Basedir support

  `config_resolver` will now search in the folders/names defined in the *XDG specification*.

1.2.4 Release 4.0.0

Features added

- Config versioning support.

  The config files can now have a section `meta` with the key `version`. The version is specified in dotted-notation with a major and minor number (f.ex.: `version=2.1`). Configuration instances take an optional `version` argument as well. If specified, config_resolver expects the `meta.version` to be there. It will raise a `config_resolver.NoVersionError` otherwise. Increments in the major number signify an incompatible change. If the application expectes a different major number than stored in the config file, it will raise a `config_resolver.IncompatibleVersion` exception. Differences in minor numbers are only logged.

Improvments

- The `mandatory` argument **has been dropped**! It is now implicitly assumed it the `.get` method does not specify a default value. Even though "explicit is better than implicit", this better reflects the behaviour of the core `ConfigParser` and is more intuitive.
- Legacy support of old environment variable names **has been dropped**!
- Python 3 support.
- When searching for a file on the current working directory, look for `./.group/app/app.ini` instead of simply `./app.ini`. This solves a conflict when two modules use config_resolver in the same application.
- Better logging.

1.2.5 Release 3.3.0

Features added

- New (optional) argument: `require_load`. If set to `True` creating a config instance will raise an error if no appropriate config file is found.
- New class: `SecuredConfig`: This class will refuse to load config files which are readable by other users than the owner.

Improvments

- Documentation updated/extended.
- Code cleanup.

1.2.6 Release 3.2.2

Improvments

- Unit tests added

1.2.7 Release 3.2.1

Fixes/Improvments

- The "group" name has been prefixed to the names of the environment variables. So, instead of APP_PATH, you can now use GROUP_APP_PATH instead. Not using the GROUP prefix will still work but emit a Deprecation-Warning.

1.2.8 Release 3.2

Features added

- The call to `get` can now take an optional default value. More details can be found in the docstring.

1.2.9 Release 3.1

Features added

- It is now possible to extend the search path by prefixing the `<APP_NAME>_PATH` variable value with a +
- Changelog added

1.3 Custom Handlers

When requesting a config-instance using `get_config()` it is possible to specify a custom *file-handler* using the `handler` keyword arg. For example:

```
from config_resolver import get_config
from config_resolver.handlers import json

result = get_config('foo', 'bar', handler=json)
```

Each handler has full control over the data type which is returned by `get_config()`. `get_config` always returns a named-tuple with two arguments:

- `config`: This contains the object returned by the `handler`.
- `meta`: This is a named-tuple which is generated by config_resolver and not modifyable by a `handler`. See *The Meta Object*.

A handler must implement the following functions/methods:

empty() → T
    This function should return an empty config instance.

from_string(*data: str*) → T
    This function should parse a string and return the corresponding config instance.

from_filename(*filename: str*) → T
    This function should open a file at *filename* and return a parsed config instance.

get_version(*instance: T*) → str
    This function should return the expected config version as string.

update_from_file(*instance: T*, *filename: str*) → None
    This function should update the existing config *instance* with the configuration found in *filename*.

    The INI and JSON handlers will update config instances by keeping the existing values and only updating the new-found values. This function could be overridden in a way that only the latest file will be kept (i.e. *replacing* configs instead of updating).

See the existing handlers in `config_resolver.handler` for some practical examples.

1.4 The Meta Object

The return value of `get_config()` returns a named-tuple which not only contains the parsed config instance, but also some additional meta-data.

Before version 5.0 this information was melded into the returned config instance.

The reason this was split this way in version 5.0, is because with this version, the return type is defined by *the handlers*. Now, handlers may have return-types which cannot easily get additional values grafted onto them (at least not explicitly). To keep it *clear and understandable*, the values are now *explicitly* returned separately! This give the handler total freedom of which data-type they work with, and still retain useful meta-data for the end-user.

The meta-object is accessible via the second return value from `get_config()`:

```
_, meta = get_config('foo', 'bar')
```

Or via the `meta` attribute on the returned named-tuple:

```
result = get_config('foo', 'bar')
meta = result.meta
```

At the time of this writing, the meta-object contains the following attributes:

active_path A list of path names were used to look for files (in order of the lookup)

loaded_files A list of filenames which have been loaded (in order of loading)

config_id The internal ID used to identify the application for which the config was requested. This corresponds to the first and second argument to `get_config`.

prefix_filter A reference to the logging-filter which was added to prefix log-lines with the config ID. This exists so a user can easily get a handle on this in case it needs to be removed from the filters.

1.5 config_resolver

1.5.1 config_resolver package

Subpackages

config_resolver.handler package

Submodules

config_resolver.handler.ini module

config_resolver.handler.json module

Module contents

Submodules

config_resolver.core module

config_resolver.dirty module

config_resolver.exc module

config_resolver.util module

Module contents

1.6 Glossary

file-handler A file-handler is a module or class offering a minimal set of functions to load files as config files. They can optionally be supplied to `get_config()`. By default, handlers for INI and JSON files are supplied. Look at *Custom Handlers* for details on how to create a new one.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index