config$_{resolver}$

Release 4.2.2

Oct 11, 2018

Contents

CHAPTER 1

Changelog

1.1 Release 4.2.5.post2

1.1.1 Fixes

- `filename` can now be passed as direct argument to `get_config`
- Don't warn if the config is retrieved correctly

1.2 Release 4.2.5.post1

1.2.1 Fixes

- Improved warning detail in deprecation messages.

1.3 Release 4.2.5

1.3.1 Fixes

- Change from a module-only distrbution to a package (for PEP-561)
- Make package PEP-561 compliant
- Add transition function `config_resolver.get_config` for a smoother upgrade to v5.0 in the future.
- Add deprecation warnings with details on how to change the code for a smooth transition to v5.0

1.4 Release 4.2.4

1.4.1 Fixes

- Improve code quality.
- Improve log message for invalid config version numbers.

1.5 Release 4.2.3

1.5.1 Fixes

- Unit tests fixed
- Added missing LICENSE file
- Log messages will now show the complete version string
- Auto-detect version number if none is specifiec in the `[meta]` section.
- Fix travis CI pipeline

1.6 Release 4.2.2

1.6.1 Fixes

- Python 2/3 class-inheritance fixed.

1.7 Release 4.2.1

1.7.1 Fixes

- Log message prefixes no longer added multiple times

1.8 Release 4.2.0

1.8.1 Features added

- Application & Group name is added to log records

1.8.2 Fixes

- Python 2/3 Unicode fix in log records

1.9 Release 4.1.0

1.9.1 Features added

- XDG Basedir support

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

1.10 Release 4.0.0

1.10.1 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.

1.10.2 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.

1.11 Release 3.3.0

1.11.1 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.

1.11.2 Improvments

- Documentation updated/extended.
- Code cleanup.

1.12 Release 3.2.2

1.12.1 Improvments

- Unit tests added

1.13 Release 3.2.1

1.13.1 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 DeprecationWarning.

1.14 Release 3.2

1.14.1 Features added

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

1.15 Release 3.1

1.15.1 Features added

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

CHAPTER 2

API

CHAPTER 3

User Manual

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

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

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

3.1 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.

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.

One last thing that `config_resolver` provides, is a better handling of default values than instances of `SafeConfigParser` of the standard library. The stdlib config parser can only specify defaults for options without associating them to a section! This means that you cannot have two options with the same name in multiple sections with different default values. `config_resolver` handles default values at the time you call `.get()`, which makes it independent of the section.

3.2 Description / Usage

The module provides two main classes:

- `Config`: This is the default class.
- `SecuredConfig`: This is a subclass of `Config` which refuses to load files which a readable by other people than the owner.

The simple usage for both is identical. The only difference is the above mentioned decision to load files or not:

```
from config_resolver imoprt Config
cfg = Config('acmecorp', 'bird_feeder')
```

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

- `/etc/acmecorp/bird_feeder/app.ini`
- `/etc/xdg/acmecorp/bird_feeder/app.ini`
- `~/.acmecorp/bird_feeder/app.ini` – This will be deprecated (no longer loaded) in `config_resolver 5.0`
- `~/.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.

3.2.1 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`

Tip: If a config file is found at `~/.<group>/<app>`, a log message with a warning is issued since config_resolver 4.1.0 encouraging the end-user to move the config file to `~/.config/<group>/<app>`.

Files are parsed using the default Python `configparser.ConfigParser` (i.e. `ini` files).

3.3 Advanced Usage

3.3.1 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 create versioned `config_resolver.Config` instances in your application:

```
cfg = 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.

3.3.2 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 `config_resolver.Config` class takes a new argument: `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 `config_resolver.Config` instance. If it's empty, nothing has been loaded.

3.4 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.

3.4.1 By the application developer

Apart from the "group name" and "application name", the `config_resolver.Config` class accepts `search_path` and `filename` as arguments. `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 = Config('acmecorp', 'bird_feeder')
db_cfg = Config('acmecorp', 'bird_feeder', filename='db.ini')
```

3.4.2 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``).

3.5 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 = Config('mycompany', 'myapplication')
```

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 instance member `_prefix_filter` if you want to change or remove it:

```
from config_resolver import Config
conf = Config('mycompany', 'myapplication')
print conf._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.

3.6 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

3.7 Difference to ConfigParser

There is one **major** difference to the default Python `ConfigParser`: the `get()` method accepts a "default" parameter. If specified, that value is returned in case `ConfigParser` does not return a value. Remember that the `ConfigParser` instance supports defaults as well if specified in the constructor.

Using the `default` parameter on `get()`, you can now have two options with the same name in two sections with *different* values. Imagine the following:

```
[database1]
dsn=sqlite:///tmp/db.sqlite3

[database2]
dsn=sqlite:///tmp/db2.sqlite3
```

In the core `ConfigParser` you could *not* specify two different default values! The `default` parameter makes this possible.

Note: *AGAIN:* The core `ConfigParser` default mechanism still takes precedence!

3.7.1 Debugging

Creating an instance of `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. For "development-time" exceptions, consider calling `get()` without a default value.

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` instance 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.

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

CHAPTER 4

Examples

A simple config instance (with logging):

```
import logging
from config_resolver import Config

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

An instance which will not load unsecured files:

```
import logging
from config_resolver import SecuredConfig

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

Loading a versioned config file:

```
import logging
from config_resolver import Config

logging.basicConfig(level=logging.DEBUG)
cfg = Config("acmecorp", "bird_feeder", version="1.0")
print cfg.get('section', 'var')
```

Default values:

```
import logging
from config_resolver import Config

logging.basicConfig(level=logging.DEBUG)
cfg = Config("acmecorp", "bird_feeder", version="1.0")
```

(continues on next page)

(continued from previous page)

```
# This will not raise an error (but emit a DEBUG log entry).
print cfg.get('section', 'example_non_existing_option_name', default=10)

# this may raise a "NoOptionError"
print cfg.get('section', 'example_non_existing_option_name')

# this may raise a "NoSectionError"
print cfg.get('example_non_existing_section_name', 'varname')
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search