Configly

Release 0.2.0

Apr 09, 2020

Contents:

CHAPTER 1

Quickstart

1.1 TL;DR

```
# config.yml
foo:
    bar: <% ENV[REQUIRED] %>
    baz: <% ENV[OPTIONAL, true] %>
list_of_stuff:
    - fun<% ENV[NICE, dament] %>al
    - fun<% ENV[AGH, er] %>al
```

```
# app.py
config = Config.from_yaml('config.yml')

print(config.foo.bar)
print(config.foo['baz'])
for item in config.list_of_stuff:
    print(item)
```

```
pip install configly[yaml]
```

1.2 Introduction

Loading configuration is done in every (application) project, and yet it is often overlooked and condidered too easy or straightforward to bother using a library to manage doing it.

Therefore, we often see code like this:

```
# config.py
import os

# Maybe it's following 12factor and loading all the config from the environment.
config = {
    'log_level': os.getenv('LOG_LEVEL'),
    'database': {
        # At least here, I can nest values if I want to organize things.
        'password': os.environ['DATABASE_PASSWORD'],
        'port': int(os.environ['DATABASE_PORT']),
    }
}
```

or this

```
# config.py
import os

class Config:
    log_level = os.getenv('LOG_LEVEL')

    # Here it's not so easy to namespace
    database_password = os.environ['DATABASE_PASSWORD']
    database_port = int(os.environ['DATABASE_PORT'])


# Oh goodness!
class DevConfig(Config):
    environment = 'dev'
```

or this

```
import configparser
# ..... Okay I dont even want to get into this one.
```

And this is all assuming that everyone is loading configuration at the outermost entrypoint! The two worst possible outcomes in configuration are:

- You are loading configuration lazily and/or deeply within your application, such that it hits a critical failure after having seemingly successfully started up.
- There is not a singular location at which you can go to see all configuration your app might possibly be reading from.

1.3 The pitch

`Configly` asserts configuration should:

- Be centralized
  - One should be able to look at one file to see all (env vars, files, etc) which must exist for the application to function.
- Be comprehensive
  - One should not find configuration being loaded secretly elsewhere

- Be declarative/static
  - code-execution (e.g. the class above) in the definition of the config inevitably makes it hard to interpret, as the config becomes more complex.
- Be namespacable
  - One should not have to prepend `foo_` namespaces to all `foo` related config names
- Be loaded, once, at app startup
  - (At least the *definition* of the configuration you're loading)
- (Ideally) have structured output
  - If something is an `int`, ideally it would be read as an int.

To that end, the `configly.Config` class exposes a series of classmethods from which your config can be loaded. It's largely unimportant what the input format is, but we started with formats that deserialize into at least `str`, `float`, `int`, `bool` and `None` types.

```
# Currently supported input formats.
config = Config.from_yaml('config.yml')
config = Config.from_json('config.json')
config = Config.from_toml('config.toml')
```

Given an input `config.yml` file:

```
# config.yml
foo:
    bar: <% ENV[REQUIRED] %>
    baz: <% ENV[OPTIONAL, true] %>
list_of_stuff:
    - fun<% ENV[NICE, dament] %>al
    - fun<% ENV[AGH, er] %>al
```

A couple of things jump out:

- Most importantly, whatever the configuration value is, it's intreted as a literal value in the format of the file which loads it. I.E. loading `"true"` from the evironment in a yaml file will yield a python `True`. Ditto `"1"`, or `"null"`.
- Each `<% ... %>` section indicates a variable
- `ENV` is an "interpolator" which knows how to obtain environment variables
- `[VAR]` Will raise an error if that piece of config is not found
- `[VAR, true]` Will `VAR` to the value after the comma
- The interpolation can be a sub-portion of a key (`fun<% ENV[NICE, dament] %>al` interpolates to "fundamental"). Another example being `'<% ENV[X, 3] %>'` interpolates to `'1'` instead of `1`

Now that you've loaded the above configuration:

```
# app.py
config = Config.from_yaml('config.yml')

# You can access namespaced config using dot access
print(config.foo.bar)

# You have use index syntax for dynamic, or non-attribute-safe key values.
print(config.foo['baz'])
```

(continues on next page)

(continued from previous page)

```
# You can iterate over lists
for item in config.list_of_stuff:
    print(item)

# You can *generally* treat key-value maps as dicts
for key, value in config.foo.items():
    print(key, value)

# You can *actually* turn key-value maps into dicts
dict(config.foo) == config.foo.to_dict()
```

1.4 Installing

```
# Basic installation
pip install configly

# To use the yaml config loader
pip install configly[yaml]

# To use the toml config loader
pip install configly[toml]

# To use the vault config loader
pip install configly[vault]
```

CHAPTER 2

Interpolators

An "interpolator" is a class which knows how to get values from a particular source by interpreting the internal portion of a dynamic config value and replacing it with a value.

Default included interpolators include:

- ENV (environment variables)
- FILE (file data)

You can use all registered interpolators when loading the configuration

```
namespace:
  env: <% ENV[ENV, production] %>
  log_level: DEBUG
  ssl_cert: FILE[ssl_cert.crt]

  theoretical_http_loaded_value: <% HTTP[localhost:5000/variable, 3] %>
```

The point is that all pieces of individual configuration can be defined centrally and declaratively, while interpolators actually go obtain that config value associate with the given input.

Configly allows the dynamic addition of new interpolator through the use of the `register_interpolator()` function.

CHAPTER 3

API

3.1 Config

class configly.**Config**(*value=None*, *_src_input=None*, *_loader=None*, *_registry=<configly.registry.Registry object>*)

Container for configuration.

```
>>> config = Config({"a": 1, "b": {"c": 2}})
>>> config.a
1
>>> config.b.c
2
```

classmethod from_json(*file*, *registry=<configly.registry.Registry object>*)

Open a toml *file* and load it into the resulting config object.

classmethod from_toml(*file*, *registry=<configly.registry.Registry object>*)

Open a toml *file* and load it into the resulting config object.

classmethod from_yaml(*file*, *registry=<configly.registry.Registry object>*)

Open a yaml *file* and load it into the resulting config object.

refresh()

Reevaluate the interpolation of variable values in the given sub-config.

This will be particularly useful for values which are coming from sources where the value might change.

to_dict()

Return a *dict* equivalent of the config object.

Roughly equivalent to >>> dict(Config({1:1})) == Config({1:1}).to_dict() True

class configly.**Interpolator**

ABC to define the interface required by an interpolator.

It is not required to subclass *Interpolator*, but it *does* provide the interface and ensures the class implements it.

__getitem__(*name*)
Override this method to implement a method to get the value for a piece of config.

This method should return a *KeyError* when the value cannot be found.

get(*name*, *default=None*)
Implement get operation with a default.

Override this method to get more tailored behavior.

class configly.**Registry**
A registry to allow for non-bundled interpolators and config loaders to be added.

By default *Config* uses a global registry, to which you can *register_interpolator*.

If you need more flexibility, you can pass *registry* to any of the *from_** classmethods to use your own registry.

```
>>> from configly import Config
>>> local_registry = Registry()
>>> config = Config.from_yaml('readthedocs.yml', registry=local_registry)
```

register_interpolator(*name*, *interpolator_cls*, *overwrite=False*)
Register a new interpolator for loading configuration from different sources.

By default *Config* classes read from a global registry of interpolators. This function registers new interpolators to that global registry.

For example, internally configly registers environment interpolation through a call like:

```
>>> from configly import EnvVarInterpolator
>>> register_interpolator("ENV", EnvVarInterpolator, overwrite=True)
```

CHAPTER 4

Contributing

4.1 Prerequisites

If you are not already familiar with Poetry, this is a poetry project, so you'll need this!

4.2 Getting Setup

See the `Makefile` for common commands, but for some basic setup:

```
# Installs the package with all the extras
make install
```

And you'll want to make sure you can run the tests and linters successfully:

```
# Runs CI-level tests, with coverage reports
make test lint
```

4.3 Need help

Submit an issue!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

C

F

G

I

R

T