Cfg-Loader

Release 0.2.0

Jun 01, 2018

Contents

Welcome to Cfg-Loader's documentation.

CHAPTER 1

User's Guide

1.1 About Cfg-Loader

Cfg-loader is a library that allows to easily setup a configuration loader that makes no assumption on the python framework you are using. It uses marshmallow to deserialize input data into a desired formatted configuration. It gives you full freedom to configure your own configuration schema.

1.1.1 Main features

- input data validation and automatic processing using marshmallow
- substitution of environment variables in input data (following docker compose variable substitution syntax)
- configuration loading from .yaml file

1.2 Quickstart

This page gives a good introduction to Cfg-Loader. If not yet install please refer to the Installation section.

Cfg-Loader is built upon marshmallow for deserializing data. It is recommended that you have some light knowledge of marshmallow before you try to setup you own configuration loader.

1.2.1 A minimal configuration loader

Declaring a configuration loader

```
>>> from cfg_loader import ConfigSchema, BaseConfigLoader
>>> from marshmallow import fields
```

(continues on next page)

(continued from previous page)

```
>>> class MyConfigSchema(ConfigSchema):
...     setting1 = fields.Str()
...     setting2 = fields.Int(required=True)
...     setting3 = fields.Float(missing=13.2)

>>> my_config_loader = BaseConfigLoader(MyConfigSchema)
```

What did we do?

1. We imported the `ConfigSchema` class, which is an enhanced version of the marshmallow base `Schema` class. `ConfigSchema` is 100% compatible with `Schema`
2. We imported `BaseConfigLoader` class which is the main class for instantiating a configuration loader
3. We imported useful marshmallow resources to declare the configuration schema.
4. We declared a configuration schema that inherits from `ConfigSchema`. This schema describes what the configuration should look like.
5. We declared a configuration loader

Loading configuration

Once a configuration loader has been declared it is possible to load configuration from objects that can be deserialized with the declared schema

```
>>> config = my_config_loader.load({
...     'setting1': 'value',
...     'setting2': '4',
... })

>>> config == {
...     'setting1': 'value',
...     'setting2': 4,
...     'setting3': 13.2,
... }
True
```

Note that `setting3` field has been automatically fulfilled in the configuration result because the field has been declared with a `missing` argument.

The same way trying to load a configuration with a required field missing is not possible

```
>>> config = my_config_loader.load({
...     'setting1': 'value',
... })
Traceback (most recent call last):
...
cfg_loader.exceptions.ValidationError: {'setting2': ['Missing data for required field.
↪']}
```

1.3 Specific features

Class `ConfigSchema` implements some specific features to make your life easier when loading a configuration from data.

1.3.1 Environment variables substitution

When loading a configuration you may like to retrieve some external information that is not directly available in your input data (typically environment variables values). Cfg-Loader allows to include placeholders in the input data that are automatically substituted with data from an external mapping at configuration loading. Cfg-Loader follows the same placeholder convention as docker compose variable substitution syntax.

Example

```
>>> substitution_mapping = {'VARIABLE': 'substitution'}
>>> my_config_loader = BaseConfigLoader(MyConfigSchema,
...                                     substitution_mapping)

>>> config = my_config_loader.load({
...     'setting1': '${VARIABLE}',
...     'setting2': '${UNSET_VARIABLE:-1}',
... })

>>> config == {
...      'setting1': 'substitution',
...      'setting2': 1,
...      'setting3': 13.2,
... }
True
```

Substitution Syntax

Syntax	Behavior
`$VARIABLE` or `${VARIABLE}`	Evaluates to `VARIABLE` value in the substitution mapping
`${VARIABLE:-default}`	Evaluates to default if `VARIABLE` is unset or empty in the substitution mapping
`${VARIABLE-default}`	Evaluates to default only if `VARIABLE` is unset in the substitution mapping
`${VARIABLE:?err}`	Exits with an error message containing `err` if `VARIABLE` is unset or empty in the substitution mapping.
`${VARIABLE?err}`	Exits with an error message containing `err` if `VARIABLE` is unset in the substitution mapping.

Substitution with shell environment variables

Typically one can choose to use `os.environ` as the substitution mapping so placeholders will be replaced with environment variables as set in the current shell.

```
>>> import os

>>> my_config_loader = BaseConfigLoader(MyConfigSchema, os.environ)
```

1.3.2 Loading configuration from .yaml file

Cfg-Loader enables you to load configuration from .yaml file

Example

tests/config/config.yml

```
base:
  name: App-Name
  path: /home/user/${PATH}

security:
  secret: $SECRET
```

Loading configuration

```
>>> from cfg_loader import YamlConfigLoader

>>> class BaseConfigSchema(ConfigSchema):
...     name = fields.Str()
...     path = fields.Str()

>>> class SecurityConfigSchema(ConfigSchema):
...     secret = fields.Str()

>>> class MyConfigSchema(ConfigSchema):
...     base = fields.Nested(BaseConfigSchema)
...     security = fields.Nested(SecurityConfigSchema)

>>> substitution_mapping = {'PATH': 'folder/file', 'SECRET': 'my-secret'}
>>> my_config_loader = YamlConfigLoader(MyConfigSchema,
...                                     substitution_mapping)

>>> config = my_config_loader.load('tests/config/config.yml')

>>> config == {
...     'base': {
...         'name': 'App-Name',
...         'path': '/home/user/folder/file',
...     },
...     'security': {
...         'secret': 'my-secret',
...     },
... }
True
```

1.3.3 Non-declared fields are preserved

If a field has been omitted when declaring a schema but this field is provided in the input data then the field will be preserved in the output configuration. No validation is performed on such a field at deserialization.

Example

```
>>> class MyConfigSchema(ConfigSchema):
...     setting1 = fields.Str()

>>> my_config_loader = BaseConfigLoader(MyConfigSchema)

>>> config = my_config_loader.load({
...     'setting1': 'value',
...     'extra': 'extra_value',
... })

>>> config == {
...     'setting1': 'value',
...     'extra': 'extra_value',
... }
True
```

1.3.4 Nested fields can be automatically unwrapped

It is sometimes useful to have a configuration schema with nested fields for better readability but you do not want your resulted configuration to have nested information. This is typically the case when you want to declare configuration by grouping settings belonging to a common family but in the end you want your configuration to have all the fields at the same level. The *UnwrapNested* field class is there for this purpose.

Example

```
>>> from cfg_loader.fields import UnwrapNested

>>> class MyNestedConfigSchema(ConfigSchema):
...     setting1 = fields.Str()
...     setting2 = fields.Int()

>>> class MyConfigSchema(ConfigSchema):
...     regular_nested = fields.Nested(MyNestedConfigSchema)
...     unwrap_nested = UnwrapNested(MyNestedConfigSchema,
...                                  prefix='my_prefix_')

>>> my_config_loader = BaseConfigLoader(MyConfigSchema)

>>> config = my_config_loader.load({
...     'regular_nested': {
...         'setting1': 'regular_value',
...         'setting2': '5',
...     },
...     'unwrap_nested': {
...         'setting1': 'unwrap_value',
...         'setting2': '4',
...     },
... })

>>> config == {
...     'regular_nested': {
...         'setting1': 'regular_value',
...         'setting2': 5,
...     },
```

(continues on next page)

(continued from previous page)

```
...     'my_prefix_setting1': 'unwrap_value',
...     'my_prefix_setting2': 4,
... }
True
```

The *UnwrapNested* inherits from Nested and can be parametrized as such.

CHAPTER 2

API Reference

2.1 API

This part of the documentation covers all the interfaces of Conf-Loader.

2.1.1 Schema

class cfg_loader.schema.base.**InterpolatingSchema**(**args*, *substitution_mapping=None*, ***kwargs*)

Schema class that interpolate environ variables from input data

It implements environment variable substitution following specification from docker-compose (c.f. https://docs.docker.com/compose/compose-file/#variable-substitution)

Parameters **substitution_mapping** – Mapping containing values to substitute

load(*data*, *many=None*, *partial=None*)

Deserialize a data structure to an object defined by this Schema's fields

Parameters

- **data** (*dict*) – Data object to load from
- **many** (*bool*) – Whether to deserialize `data` as a collection
- **partial** (*bool | tuple*) – whether to ignore missing fields

Returns Deserialized data

class cfg_loader.schema.base.**ExtraFieldsSchema**(*only=None*, *exclude=()*, *prefix=''*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Schema class that preserves fields provided in input data but that were omitted in schema fields

add_extra_fields(*data*, *original_data*)

Add field from input data that were not listed as a schema fields

Parameters

- **data** (*dict*) – Data to complete
- **extra_data** (*dict*) – Extra data to insert

class cfg_loader.schema.base.**UnwrapNestedSchema**(*only=None*, *exclude=()*, *prefix=''*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Schema class that can unwrap nested fields

class cfg_loader.schema.base.**ConfigSchema**(**args*, *substitution_mapping=None*, ***kwargs*)

Main schema class for declaring a configuration schema

It inherits every feature from

- *InterpolatingSchema*
- *ExtraFieldsSchema*
- *UnwrapNestedSchema*

Example

```
>>> from cfg_loader import ConfigSchema, BaseConfigLoader
>>> from marshmallow import fields
```

```
>>> class MyConfigSchema(ConfigSchema):
...     setting1 = fields.Str()
...     setting2 = fields.Int(required=True)
...     setting3 = fields.Float(missing=13.2)
```

```
>>> substitution_mapping = {'VARIABLE': 'substitution'}
>>> my_config_loader = BaseConfigLoader(MyConfigSchema,
...                                     substitution_mapping)
```

```
>>> schema = MyConfigSchema(substitution_mapping=substitution_mapping)
>>> config = schema.load({
...    'setting1': '${VARIABLE}',
...    'setting2': '${UNSET_VARIABLE:-1}',
... })
```

```
>>> config == {
...     'setting1': 'substitution',
...     'setting2': 1,
...     'setting3': 13.2,
... }
True
```

2.1.2 Loader

class cfg_loader.loader.**BaseConfigLoader**(*config_schema*, *substitution_mapping=None*)

Base config loader using a marshmallow schema to validate and process input data

Parameters **schema** – Marshmallow schema used to deserialize configuration input data

load(*data*, *substitution_mapping=None*)

Load configuration from an object

Parameters **input** (*object*) – Data to load configuration from (must be deserializable by schema)

class cfg_loader.loader.**YamlConfigLoader**(**args*, *config_file_env_var='CONFIG_FILE'*, *default_config_path=None*, ***kwargs*)

Config loader that reads config from .yaml file

Parameters

- **config_file_env_var** (*str*) – Environment variable to read config file path from if not provided when loading
- **default_config_path** (*str*) – Used if neither path is provided at loading nor environment variable

check_file(*config_file*)

Check file validity

Parameters **config_file** (*str*) – Path to the .yaml configuration file

load(*config_file=None*, *substitution_mapping=None*)

Load configuration from .yaml file

Parameters **config_file** (*str*) – Path to the .yaml configuration file

2.1.3 Interpolator

class cfg_loader.interpolator.**Interpolator**(*substitution_mapping=None*, *substitution_template=<class 'cfg_loader.interpolator.SubstitutionTemplate'>*)

Class used to substitute environment variables in complex object

Parameters **substitution_mapping** (*dict*) – Mapping with values to substitute

Example

```
>>> interpolator = Interpolator(substitution_mapping={'VARIABLE': 'value'})
```

```
>>> interpolator.interpolate('${VARIABLE} in complex string')
'value in complex string'
```

```
>>> result = interpolator.interpolate_recursive({'key1': '${VARIABLE}', 'key2': [
↪'element', '${EXTRA-default}']})
>>> result == {'key1': 'value', 'key2': ['element', 'default']}
True
```

interpolate(*string*)

Substitute environment variable in a string

interpolate_recursive(*obj*)

Substitute environment variable in an object

class cfg_loader.interpolator.**SubstitutionTemplate**(*template*)

Class used to substitute environment variables in a string

It implements specification from docker-compose environ variable substitution (c.f. https://docs.docker.com/compose/compose-file/#variable-substitution)

Examples with basic substitution

```
>>> template = SubstitutionTemplate('${VARIABLE}')
>>> template.substitute({'VARIABLE': 'value'})
'value'
>>> template.substitute({'VARIABLE': ''})
''
>>> template.substitute({})
Traceback (most recent call last):
...
KeyError: 'VARIABLE'
```

Examples with substitution if variable is empty or unset (separator: ":-")

```
>>> template = SubstitutionTemplate('${VARIABLE:-default}')
>>> template.substitute({'VARIABLE': 'value'})
'value'
>>> template.substitute({'VARIABLE': ''})
'default'
>>> template.substitute({})
'default'
```

Examples with substitution if variable is empty (separator: "-"):

```
>>> template = SubstitutionTemplate('${VARIABLE-default}')
>>> template.substitute({'VARIABLE': 'value'})
'value'
>>> template.substitute({'VARIABLE': ''})
''
>>> template.substitute({})
'default'
```

Examples with error raised if variable is unset (separator: "?")

```
>>> template = SubstitutionTemplate('${VARIABLE?err}')
>>> template.substitute({'VARIABLE': 'value'})
'value'
>>> template.substitute({'VARIABLE': ''})
''
>>> template.substitute({})
Traceback (most recent call last):
...
cfg_loader.exceptions.UnsetRequiredSubstitution: err
```

Examples with error raised if variable is empty or unset (separator: ":?")

```
>>> template = SubstitutionTemplate('${VARIABLE:?err}')
>>> template.substitute({'VARIABLE': 'value'})
'value'
>>> template.substitute({'VARIABLE': ''})
Traceback (most recent call last):
...
cfg_loader.exceptions.UnsetRequiredSubstitution: err
>>> template.substitute({})
Traceback (most recent call last):
...
cfg_loader.exceptions.UnsetRequiredSubstitution: err
```

substitute(*mapping*)

Substitute values indexed by mapping into *template*

Parameters **mapping** (*dict*) – Mapping containing values to substitute

2.1.4 Fields

class cfg_loader.fields.**Path**(**args*, ***kwargs*)
A validated path field. Validation occurs during both serialization and deserialization.

Parameters

- **args** – The same positional arguments that String receives.
- **kwargs** – The same keyword arguments that String receives.

class cfg_loader.fields.**UnwrapNested**(**args*, *prefix=''*, ***kwargs*)
Nested fields class that will unwrapped at deserialization

Useful when used with UnwrapSchema

Parameters **prefix** (*str*) – Optional prefix to add to every key when unwrapping a field

2.1.5 Exceptions

class cfg_loader.exceptions.**ConfigLoaderError**
Bases: Exception

Base class for all application related errors.

class cfg_loader.exceptions.**ConfigFileMissingError**
Bases: *cfg_loader.exceptions.ConfigLoaderError*

Error raised when not specifying a config file

class cfg_loader.exceptions.**ConfigFileNotFoundError**
Bases: *cfg_loader.exceptions.ConfigLoaderError*

Error raised when an invalid config file path is provided

class cfg_loader.exceptions.**LoadingError**
Bases: *cfg_loader.exceptions.ConfigLoaderError*

Error raised when loading the configuration file

class cfg_loader.exceptions.**ValidationError**
Bases: *cfg_loader.exceptions.ConfigLoaderError*

Error raised when marshmallow raise a validation error at deserialization

class cfg_loader.exceptions.**UnsetRequiredSubstitution**
Bases: *cfg_loader.exceptions.LoadingError*

Error raised when a substitution environment variable is required but unset

class cfg_loader.exceptions.**InvalidSubstitution**
Bases: *cfg_loader.exceptions.LoadingError*

Error raised when an invalid substitution is detected

CHAPTER 3

Contributing

If you are interested in contributing to the project please refer to *Contributing guidelines*

3.1 Contributing guidelines

3.1.1 Feature Requests, Bug Reports, and Feedback...

...should all be reported on the GitHub Issue Tracker .

Reporting issues

- Describe what you expected to happen.
- If possible, include a minimal, complete, and verifiable example to help
- Describe what actually happened. Include the full traceback if there was an exception.

3.1.2 Setting-Up environment

Requirements

1. Having the latest version of `git` installed locally
2. Having Python 3.6 installed locally
3. Having `virtualenv` installed locally

   To install `virtualenv` you can run the following command

   ```
   $ pip install virtualenv
   ```

4. Having `docker` and `docker-compose` installed locally

5. Having `pip` environment variables correctly configured

   Some of the package's dependencies of the project could be hosted on a custom PyPi server. In this case you need to set some environment variables in order to make `pip` inspect the custom pypi server when installing packages.

   To set `pip` environment variables on a permanent basis you can add the following lines at the end of your `\.bashrc` file (being careful to replace placeholders)

```
# ~/.bashrc

...

# Indicate to pip which pypi server to download from
export PIP_TIMEOUT=60
export PIP_INDEX_URL=<custom_pypi_protocol>://<user>:<password>@<custom_pypi_host>
export PIP_EXTRA_INDEX_URL=https://pypi.python.org/simple
```

First time setup

- Clone the project locally
- Create development environment using Docker or Make

```
$ make init
```

3.1.3 Project organisation

The project

```
.
├── cfg_loader/           # Main package source scripts (where all functional python␣
↪scripts are stored)
├── docs/                 # Docs module containing all scripts required by sphinx␣
↪to build the documentation
├── tests/                # Tests folder where all test modules are stores
├── .coveragerc           # Configuration file for coverage
├── .gitignore            # List all files pattern excluded from git's tracking
├── .gitlab-ci.yml        # GitLab CI script
├── AUTHORS               # List of authors of the project
├── CHANGES               # Changelog listing every changes from a release to␣
↪another
├── CONTRIBUTING.rst      # Indicate the guidelines that should be respected when␣
↪contributing on this project
├── LICENSE               # License of the project
├── Makefile              # Script implement multiple commands to facilitate␣
↪developments
├── README.rst            # README.md of your project
├── setup.cfg             # Configuration of extra commands that will be installed␣
↪on package setup
├── setup.py              # File used to setup the package
└── tox.ini               # Configuration file of test suite (it runs test suite␣
↪in both Python 3.5 and 3.6 environments)
```

3.1.4 Coding

Development Workflow

Please follow the next workflow when developing

- Create a branch to identify the feature or issue you will work on (e.g. `feature/my-feature` or `hotfix/2287`)
- Using your favorite editor, make your changes, committing as you go and respecting the AngularJS Commit Message Conventions
- Follow PEP8 and limit script's line length to **120 characters**. See *testing-linting*
- Include tests that cover any code changes you make. See *running-test* and *running-coverage*
- Update `setup.py` script with all dependencies you introduce. See *adding-dependency* for precisions
- Write clear and exhaustive docstrings. Write docs to precise how to use the functionality you implement. See *writing-docs*
- Update changelog with the modifications you proceed to. See *updating-changelog*
- Your branch will soon be merged ! :-)

Testing

Running tests

Run test suite in by running

```
$ make test
```

Running coverage

Please ensure that all the lines of source code you are writing are covered in your test suite. To generate the coverage report, please run

```
$ make coverage
```

Read more about coverage.

Running the full test suite with `tox` will combine the coverage reports from all runs.

Testing linting

To test if your project is compliant with linting rules run

```
$ make test-lint
```

To automatically correct linting errors run

```
$ make lint
```

Running full test suite

Run test suite in multiple distinct python environment with following command

```
$ make tox
```

Writing documentation

Write clear and exhaustive docstrings in every functional scripts.

This project uses sphinx to build documentations, it requires docs file to be written in `.rst` format.

To build the documentation, please run

```
$ make docs
```

Precisions

Updating changelog

Every implemented modifications on the project from a release to another should be documented in the changelog `CHANGES.rst` file.

The format used for a release block is be the following

```
Version <NEW_VERSION>
---------------------

Released on <NEW_VERSION_RELEASED_DATE>, codename <NEW_VERSION_CODENAME>.

Features

- Feature 1
- Feature 2
- Feature 3

Fixes

- Hotfix 1 (``#134``)
- Hotfix 2 (``#139``)

.. _#134: https://github.com/nmvalera/cfg-loader/issues/134
.. _#139: https://github.com/nmvalera/sandbox/cfg-loader/issues/139
```

Be careful to never touch the header line as well as the release's metadata sentence.

```
Version <NEW_VERSION>
---------------------

Released on <NEW_VERSION_RELEASED_DATE>, codename <NEW_VERSION_CODENAME>.
```

Adding a new dependency

When adding a new package dependency it should be added in `setup.py` file in the `install_requires` list

The format should be `dependency==1.3.2`.

When adding a dev dependency (e.g. a testing dependency) it should be added in

- `setup.py` file in the `extra_requires` `dev` list
- `tox.ini` file in the `[testenv]` `deps`

3.1.5 Makefile commands

`Makefile` implements multiple handful shell commands for development

make init

Initialize development environment including

- venv creation
- package installation in dev mode

make clean

Clean the package project by removing some files such as `.pyc`, `.pyo`, `*.egg-info`

make test-lint

Check if python scripts are compliant with PEP8 rules

make lint

Automatically correct PEP8 mistakes contained in the project.

make coverage

Run the test suite and computes test coverage. It creates an html report that is automatically open after the commands terminates

make tox

Run the test suites in multiple environments

make docs

Build documentation from the `docs` folder using sphinx. It generates a build of the documentation in html format located in `docs/_build/html`.

CHAPTER 4

Additional Notes

Legal information and changelog are here for the interested.

4.1 Changelog

Here you can see the full list of changes between each releases of Cfg-Loader.

4.1.1 Version 0.2.0

Released on June 1st 2018

Feature

- implement default config path

4.1.2 Version 0.1.3

Released on May 30th 2018

Docs

- setup readthedocs

4.1.3 Version 0.1.2

Released on May 30th 2018

Docs

- fix badges in README

4.1.4 Version 0.1.1

Released on May 30th 2018

No Change

4.1.5 Version 0.1.0

Released on May 30th 2018

Features

- input data validation and automatic processing using marshmallow
- substitution of environment variables in input data (following docker compose variable substitution syntax)
- configuration loading from .yaml file

4.1.6 Version 0.0.0

Unreleased

Chore

- Project: Initialize project

4.2 License

4.2.1 Authors

Cfg-Loader is developed and maintained by the ConsenSys France team and community contributors. The core maintainers are:

- Nicolas Maurice (nmvalera)

4.2.2 General License Definitions

The following section contains the full license texts for Cfg-Loader and the documentation.

- "AUTHORS" hereby refers to all the authors listed in the *Authors* section.
- The "*License*" applies to all the source code shipped as part of Cfg-Loader (Cfg-Loader itself as well as the examples and the unit tests) as well as documentation.

4.2.3 License

Redistribution and use in source and binary forms of the software as well as documentation, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Config-Loader nor the names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index