django-confit Documentation

Release 0.3

Benoît Bryon

March 23, 2015

Contents

1 Example **3**

2 Project status **5**

3 Ressources **7**

4 Contents **9**

4.1 Install 9
4.2 Configure 10
4.3 Load configuration 11
4.4 Validate schemas 12
4.5 Write configuration schemas 12
4.6 Builtin schemas 13
4.7 Demo project 14
4.8 About django-confit 14
4.9 Contributing to django-confit 16

5 Indices and tables **19**

Python Module Index **21**

django-confit eases Django configuration management.

As a Django user, in order to configure a project:

- *django-confit* helps you load the settings wherever they are, whatever the format: Python modules, environment variables, JSON, YAML...
- *django-confit* validates the settings, i.e. it tells you if some directive is missing, has wrong format...

As a Django library developer, in order to help your application's users:

- you write configuration schemas for your application, using *django-confit*'s toolkit and conventions.
- *django-confit* helps you document your application's specific configuration.

As a non Django user, in order to deploy and run a Django-powered project:

- you write the configuration as you like, depending on your workflow and your provisioning toolkit. You know the project can load them using *django-confit*.
- you expect applications to validate the configuration before they actually use it, and report errors with a readable output.

CHAPTER 1

Example

In a project's `settings.py` file, let's load configuration from various locations, then validate it:

```python
import os

import django_confit

# Load settings.
raw_settings = {}
raw_settings.update(django_confit.load_module('myproject.default_settings'))
raw_settings.update(django_confit.load_file(open('/etc/myproject.json')))
raw_settings.update(django_confit.load_mapping(os.environ, prefix='MYPROJECT_'))

# Validate and clean settings.
cleaned_settings = django_confit.validate_settings(raw_settings)

# Update globals, because that's the way Django uses DJANGO_SETTINGS_MODULE.
globals().update(cleaned_settings)
```

CHAPTER 2

Project status

Today, *django-confit* is a proof of concept:

- loading settings is nice and easy.
- validating configuration is easy... provided you have the schemas.
- creating configuration schemas is verbose. It uses colander [1] which has nice features, but may not be the definitive option.
- generating documentation from schemas is not implemented.

The main limitation is that schemas are mandatory. If some configuration directive is not registered in a schema, it will not be present in validation output. It means that, if you install a new third-party Django application, you need the configuration schema for this application, else its settings will not pass validation. **So the most-wanted contribution is submitting configuration schemas for third-party applications.**

Notice that this behaviour is a wanted feature. As *django-confit* author, I think **libraries should always provide a schema for the settings they use**. I do not pretend *django-confit* should be THE answer. I just bet that, if schemas were widely adopted by the Django community, configuration would be easier to manage.

django-confit does not pretend to be the ultimate configuration management app for Django. Its goal is to show how some issues could be resolved, and to highlight the benefits. *django-confit* is a proposal. If you like its concepts, then you can:

- use *django-confit* of course!
- discuss, spread the word, send feedback.
- improve code. Help around configuration schemas of third-party apps would be appreciated.

[1] https://pypi.python.org/pypi/colander/

CHAPTER 3

Ressources

- Documentation: https://django-confit.readthedocs.org
- PyPI page: https://pypi.python.org/pypi/django-confit/
- Code repository: https://github.com/benoitbryon/django-confit
- Bugtracker: https://github.com/benoitbryon/django-confit/issues
- Continuous integration: https://travis-ci.org/benoitbryon/django-confit
- Roadmap: https://github.com/benoitbryon/django-confit/milestones

CHAPTER 4

Contents

4.1 Install

django-confit is open-source, published under BSD license. See *License* for details.

If you want to install a development environment, you should go to *Contributing to django-confit* documentation.

4.1.1 Prerequisites

- Python [1] 2.7, 3.3

4.1.2 As a library

In most cases, you will use *django-confit* as a dependency of another project (typically a Django project or a Django application). In such a case, you should add `django-confit` in your main project's requirements. Typically in `setup.py`:

```
from setuptools import setup

setup(
    install_requires=[
        'django-confit',
        #...
    ]
    # ...
)
```

Then when you install your main project with your favorite package manager (like pip [2]), *django-confit* will automatically be installed.

4.1.3 Standalone

You can install *django-confit* with your favorite Python package manager. As an example with pip [2]:

```
pip install django-confit
```

[1] https://www.python.org/
[2] https://pypi.python.org/pypi/pip/

4.1.4 Check

Check *django-confit* has been installed:

```
python -c "import django_confit;print(django_confit.__version__)"
```

You should get *django_confit*'s version.

Notes & references

See also:

Changelog

4.2 Configure

Once *django-confit* is *installed*, let's configure it. This section describes directives related to *django-confit* in your Django settings.

4.2.1 CONFIT_SCHEMAS

A dictionary of `<APP>: <SCHEMA>` where `<APP>` is in `INSTALLED_APPS` and `<SCHEMA>` is the Python dotted path to a Schema class (or factory).

This configuration directive is mandatory **only when** you need configuration schemas that are not builtin *django-confit*. See also *the list of builtin schemas*.

`validate_settings()` automatically loads schemas from `INSTALLED_APPS` setting. For each *<APP>* in `INSTALLED_APPS`, it looks for:

1. `CONFIT_SCHEMAS[<APP>]`,
2. `django_confit.schemas.<APP>.ConfigurationSchema`.

Simple example from default settings in django-confit demo project [3]:

```
CONFIT_SCHEMAS = {
    'django_confit_demo': 'django_confit_demo.settings_schemas'
                          '.DjangoConfitDemoConfigurationSchema'
}
```

Note: Early versions of *django_confit* tried to automatically load schemas, without having to register them:

1. `settings_schemas.<APP>.ConfigurationSchema` in current package, to allow local overrides
2. `<APP>.settings_schemas.ConfigurationSchema`, to allow third-party applications to manage their own schema.
3. `django_confit.schemas.<APP>.ConfigurationSchema` to load *django-confit*'s builtins.

The idea was nice. But it did not work. Because while trying to import `<APP>.settings_schemas.ConfigurationSchema`, if `import <APP>` imports Django stuff, then a circular import can occur. That is a pain to debug and a pain to fix.

With a registry, we import only things that are expected to work.

[3] https://github.com/benoitbryon/django-confit/blob/master/demo/django_confit_demo/default_settings.py

Notes & references

4.3 Load configuration

As a Django user, you manage configuration as you like: as Python modules, as JSON or YAML files, as environment variables... Do whatever integrates best with your deployment workflow and your team.

As a non Django user (such as a member of the Ops team), you may appreciate putting configuration in configuration files or in environment variables. Not in Python code.

This document explains how *django-confit* can help Django users load configuration from various locations and formats.

4.3.1 load_mapping

`django_confit.loaders.`**`load_mapping`**`(`*input*, *prefix=''*`)`

Convert mapping of {key: string} to {key: complex type}.

This function makes it possible (and easy) to load complex types from single-level key-value stores, such as environment variables or INI files.

Of course, both flat and nested mappings are supported:

```
>>> from django_confit import load_mapping
>>> flat_mapping = {'DEBUG': 'True', 'SECRET_KEY': 'not a secret'}
>>> output = load_mapping(flat_mapping)
>>> output == flat_mapping
True

>>> nested_mapping = {'DATABASES': {'USER': 'me', 'HOST': 'localhost'}}
>>> output = load_mapping(nested_mapping)
>>> output == nested_mapping
True
```

Values can be complex types (sequences, mappings) using JSON or YAML. Keys using ".json" or ".yaml" suffix are automatically decoded:

```
>>> nested_mapping = {
...     'DATABASES.yaml': 'ENGINE: sqlite3',
... }
>>> output = load_mapping(nested_mapping)
>>> output['DATABASES'] == {'ENGINE': 'sqlite3'}
True
```

You can use optional `prefix` argument to load only a subset of mapping:

```
>>> mapping = {'YES_ONE': '1', 'NO_TWO': '2'}
>>> load_mapping(mapping, prefix='YES_')
{'ONE': '1'}
```

4.3.2 load_module

`django_confit.loaders.`**`load_module`**`(`*module_path*`)`

Return module's globals as a dict.

```
>>> from django_confit import load_module
>>> settings = load_module('django.conf.global_settings')
>>> settings['DATABASES']
{}
```

It does not load "protected" and "private" attributes (those with underscores).

```
>>> '__name__' in settings
False
```

4.3.3 load_file

`django_confit.loaders.load_file(file_obj)`

Return mapping from file object, using `name` attr to guess format.

Supported file formats are JSON and YAML. The lowercase extension is used to guess the file type.

```
>>> from django_confit import load_file
>>> from six.moves import StringIO
>>> file_obj = StringIO('SOME_LIST: [a, b, c]')
>>> file_obj.name = 'something.yaml'
>>> load_file(file_obj) == {
...     'SOME_LIST': ['a', 'b', 'c'],
... }
True
```

4.4 Validate schemas

As a Django user, once you have *loaded configuration*, you want to make sure it is valid. It is a must-have feature when the configuration comes from various locations. Or when it has been written by several users.

As a non Django user (such as a member of the Ops team, who deploys and configure services), you do **not** want to be prosecuted because you made a typo in configuration. You think applications should validate the configuration before they actually use it, and report errors with a readable output.

This section explains how *django-confit* helps Django users validate configuration.

4.4.1 validate_settings

`django_confit.schemas.validate_settings(raw_settings)`

Return cleaned settings using schemas collected from INSTALLED_APPS.

4.4.2 Low-level API

Use `deserialize()` method of *Colander* schemas: http://docs.pylonsproject.org/projects/colander/en/latest/basics.html#deserializati

4.5 Write configuration schemas

As a Django application developer, you want to expose configuration directives. You want to make sure your application is well configured. As an example, you appreciate to mark some directives as required, and to provide default values for optional ones. But you do not want to re-invent the wheel about validation.

As a Django user, you do not want to bother about application configuration schemas. You just want it loaded when you register the application in `INSTALLED_APPS`.

This sections explains how to create with *django-confit*.

4.5.1 Create settings_schemas.ConfigurationSchema

See:

- colander's documentation [4]
- django-confit's builtin schemas [5]

Simple example from custom schemas in django-confit demo project [6]:

```
"""Custom configuration schema(s)."""
import colander


class DjangoConfitDemoConfigurationSchema(colander.MappingSchema):
    CONFITDEMO_HELLO = colander.SchemaNode(
        colander.String(),
        missing=colander.required,
        default='',
    )
```

Warning: In *django-confit* code, you will see some hacks around versions of applications. Yes, they are **hacks**. They are necessary because from the *django-confit* point of view, we do not know which version of the application is actually installed. So we have to write conditionals in order to support them as well as possible... But it is a true pain.
Just imagine the schema lives within the third-party application itself... As a consequence:

- one schema is enough. No need to manage concurrent schemas. The one for the current version is enough.
- the schema is updated along the application. It is part of the application's release process.
- as an user, you do not have to find or write a schema for the application. There is an official one! You install the app, you get the schema!

4.5.2 Register your schema

In a project or a third-party application, register schemas in `settings.CONFIT_SCHEMAS`. See *Configure*.

In *django-confit* itself, make sure the schema can be loaded with `django_confit.schemas.<APP>.ConfigurationSchema`.

Notes & references

4.6 Builtin schemas

Here is the list of configuration schemas implemented as part of *django-confit*. They are schemas that will automatically be loaded.

- Django 1.5 and Django 1.6 are supported

[4] http://docs.pylonsproject.org/projects/colander
[5] https://github.com/benoitbryon/django-confit/tree/master/django_confit/schemas
[6] https://github.com/benoitbryon/django-confit/blob/master/demo/django_confit_demo/settings_schemas.py

- django-debug-toolbar
- django-nose
- django-pimpmytheme
- django-pipeline
- raven

Note: Maintaining schemas for third-party applications is hard! Any help would be welcome, as an example:

- report bugs ;
- improve existing schemas (smarter validation) ;
- watch upgrades, support several versions.

If you are a library developer and you like *django-confit*, the best thing you can do is to distribute your app's schema inside your app! So that you maintain the schema along with your application, and *django-confit* does not have to deal with multiple versions (given a version of your app, there is only one valid schema for your app, and it lives within your app).

4.7 Demo project

Demo folder in project's repository [7] contains a Django project to illustrate *django-confit* usage.

Notes & references

4.8 About django-confit

This section is about *django-confit* project itself.

4.8.1 Vision

django-confit has been created with the following concepts in mind:

- Local configuration should not live in code. Altering `sys.path` or putting local settings module within code is not a good practice. Moreover, code should not vary from one environment to another.
- Environment variables are not enough to manage settings. You can load settings from environment variables, but you can also do it from files. Files have many benefits (backup, version control...).
- Settings imported from external locations should (always) be validated.
- Application should expose the full list of settings they use.
- Documentation about applications settings should be auto-generated.

[7] https://github.com/benoitbryon/django-confit/tree/master/demo/

4.8.2 Alternatives and related projects

This document presents other projects that provide similar or complementary functionalities. It focuses on differences with django-confit.

As a matter of fact, there are many projects related to configuration on PyPI [8] (as examples, search for `config` or `django conf`).

django-configglue

django-configglue [9] inspired *django-confit*: it covers both loading and validation of settings. *django-configglue* is tied to INI files. The schemas look less powerful than those provided by colander [10].

ConfigIt

ConfigIt [11] can load settings from various formats.

References

4.8.3 License

Copyright (c) 2014, Benoît Bryon. All rights reserved.

Redistribution and use in source and binary forms, 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 django-confit nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

[8] https://pypi.python.org/pypi
[9] https://pypi.python.org/pypi/django-configglue
[10] https://pypi.python.org/pypi/colander/
[11] https://pypi.python.org/pypi/ConfigIt/

4.8.4 Authors & contributors

Maintainer: Benoît Bryon <benoit@marmelune.net>

Developers: https://github.com/benoitbryon/django-confit/graphs/contributors

4.8.5 Changelog

This document describes changes between past releases. For information about future releases, check milestones [12] and *Vision*.

0.3 (2015-03-23)

Upgrades: Django and Sphinx.

- Feature #26 - Added support for Django up to versions 1.5.12 and 1.6.11.
- Bug #25 - Fixed support of latest Sphinx version.

0.2 (2014-06-30)

- Feature #13 - Extended support for Django from 1.5 to 1.5.8 and from 1.6 to 1.6.5.
- Feature #16 - Added schema for django-pimpmytheme settings.
- Bug #15 - DjangoSettingsSchema.TEMPLATE_LOADERS now accepts either strings or tuples (was "accepts only strings"), as specified in Django's settings reference documentation.

0.1 (2014-04-21)

Proof of concept.

- Introduced configuration schemas, using Colander.
- Implemented configuration schema for Django 1.5.5 and Django 1.6.2.
- Implemented configuration schema for some third-party libraries: django-pipeline, django-debug-toolbar, django-nose and raven.
- Introduced configuration loading utilities.
- Feature #7 - Code repository contains a demo project, used for documentation and tests.
- Feature #9 - Demo project shows how to register a custom configuration schema.
- Feature #10 - Validate_settings() emits a warning if some directives used in raw settings input are not in cleaned settings output. Helps figure out which schemas are missing.

Notes & references

4.9 Contributing to django-confit

This document provides guidelines for people who want to contribute to the project.

[12] https://github.com/benoitbryon/django-confit/milestones

4.9.1 Create tickets

Please use django-confit bugtracker [13] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, so that we are given an opportunity to give feedback as soon as possible.
- Then, in your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

4.9.2 Use topic branches

- Work in branches.
- Prefix your branch with the ticket ID corresponding to the issue. As an example, if you are working on ticket #23 which is about contribute documentation, name your branch like `23-contribute-doc`.
- If you work in a development branch and want to refresh it with changes from master, please rebase [14] or merge-based rebase [15], i.e. do not merge master.

4.9.3 Fork, clone

Clone *django-confit* repository (adapt to use your own fork):

```
git clone git@github.com:benoitbryon/django-confit.git
cd django-confit/
```

4.9.4 Usual actions

The *Makefile* is the reference card for usual actions in development environment:

- Install development toolkit with pip [16]: `make develop`.
- Run tests with tox [17]: `make test`.
- Build documentation: `make documentation`. It builds Sphinx [18] documentation in *var/docs/html/index.html*.
- Release *django-confit* project with zest.releaser [19]: `make release`.
- Cleanup local repository: `make clean`, `make distclean` and `make maintainer-clean`.

See also `make help`.

Notes & references

[13] https://github.com/benoitbryon/django-confit/issues
[14] http://git-scm.com/book/en/v2/Git-Branching-Rebasing
[15] http://tech.novapost.fr/psycho-rebasing-en.html
[16] https://pypi.python.org/pypi/pip/
[17] https://pypi.python.org/pypi/tox/
[18] https://pypi.python.org/pypi/Sphinx/
[19] https://pypi.python.org/pypi/zest.releaser/

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

Notes & references

d

django_confit.loaders, 11

Index

D

django_confit.loaders (module), 11

L

load_file() (in module django_confit.loaders), 12
load_mapping() (in module django_confit.loaders), 11
load_module() (in module django_confit.loaders), 11

V

validate_settings() (in module django_confit.schemas), 12