Django YAMLCONF Documentation

Release 1.4.0

Michael Rohan <mrohan@vmware.com>

Aug 25, 2023

Contents

`django_yamlconf` is part of VMware's support of open source development and community.

Handle YAML based Django settings: load Django settings from YAML files based on a Django project name. The YAML files loaded start with a YAML file in the directory containing the Django settings file and then loads any other YAMLCONF files up the directory tree from the initial file. Values from files higher up the directory tree over-ride lower in the tree. The contents of the YAML file simply defines values that over-ride (or add to) attributes of the standard Django settings file, e.g., for the project "buildaudit", the settings.py file could contain:

```
DEBUG = True
```

i.e., the value for development. This can be redefined via a `buildaudit.yaml` file using the definition:

```
DEBUG: false
```

If the environment variable `YAMLCONF_CONFFILE` is defined, it uses as the final YAML file loaded (in this case, the file name does not need to match the project name and it can be located anywhere in the file system).

CHAPTER 1

Quick Start

The YAMLCONF definitions are added to the Django settings file by including a call to the `load` function in the settings file. This would normally be towards the end of the settings file. The simplest, and likely normal usage is to call without arguments. YAMLCONF will infer the project information from the call stack. For a standard Django application structure, the settings file:

```
myproject/myproject/settings.py
```

would contain the development oriented definitions, e.g., database definitions for user and password for a development database. The settings file would then end with a call the the `load` function. Additional definitions could be defined after the `load` function to update conditional definitions, e.g., if `DEBUG` is enabled.

```
import django_yamlconf

...

DATABASES = {
    'default': {
        'NAME': 'example',
        'USER': 'example',
        'PASSWORD': 'example',
        'HOST': 'localhost',
        ...
    }
}
...

django_yamlconf.load()
```

On a production server, for this example, a `myproject.yaml` would be put in place containing the host name for the production database and the password for the example user (assuming production is using the same database name and username). In this example, a random `pwgen` password is used:

```
DATABASES.default.PASSWORD: 'zibiemohjuD6foh0'
DATABASES.default.HOST: 'myproject-db.eng.vmware.com'
```

See the `load` function for more information on other optional arguments.

CHAPTER 2

License

`django-yamlconf` is release under the BSD-2 license, see the LICENSE file.

SPDX-License-Identifier: BSD-2-Clause

Contents:

2.1 Management Commands

YAMLCONF includes three management commands (`django_yamlconf` needs to be added to the `INSTALLED_APPS` to add these commands):

- `ycexplain`: explain where an attribute value was defined
- `yclist`: list the attribute values defined via YAMLCONF
- `ycsysfiles`: Create system control files based on attribute controlled template files

The attributes available to the management commands can be extended using methods returning dictionaries of values. The method names can be defined in the `settings` file or via a YAMLCONF file via the attribute `YAMLCONF_ATTRIBUTE_FUNCTIONS`, e.g.,

```
YAMLCONF_ATTRIBUTE_FUNCTIONS:
  - 'health_checks.ycattrs.attributes'
```

As can be seen from the example method above, these additional attibutes are primarily used with the `ycsysfiles` command.

2.1.1 `ycexplain` Command

This `ycexplain` gives information on the value defined by the set of YAML files loaded for an application along with any documentation and information on eclipsed attribute values lower in the directory tree structure. For example, for the `DEBUG` attribute:

```
$ python manage.py ycexplain DEBUG
---------------------------
DEBUG = "False" (via "/u/mrohan/clients/xmpl/buildaudit.yaml")

Documentation:
    Enable or disable debugging functionality.  On the production
    server this attribute should be set to false

Eclipsed values:
    "True" via "/u/mrohan/clients/xmpl/buildaudit/buildaudit.yaml"
    "True" via "buildaudit.settings"
```

2.1.2 `yclist` Command

The `yclist` command simply lists the attributes defined via YAML files, e.g.,:

```
$ python manage.py yclist
Listing YAMLCONF managed attributes

ALLOWED_HOSTS                      ['localhost']
BACKUP_CONFIG.directory            {BASE_DIR}/backup
BASE_DIR                           /home/mrohan/clients/osstp-yc/webapps
CONTROL_FILE                       {WEBAPPS_DIR}/osstpmgt.yaml
DATABASES.default.CONN_MAX_AGE     600
DATABASES.default.HOST             {DBHOST}
DATABASES.default.NAME             {DBNAME}
DATABASES.default.PASSWORD         {DBPASSWORD}
DATABASES.default.USER             {DBUSER}
DBHOST                             localhost
DBNAME                             osstp
DBPASSWORD                         A-Password
DBUSER                             osstp
INSTALL_DIR                        /var/oss/osstp
MANAGE_PY                          {WEBAPPS_DIR}/manage.py
OS_MACHINE                         x86_64
OS_NODE                            mrohan-osstp-yc
OS_PROCESSOR                       x86_64
OS_RELEASE                         4.4.0-101-generic
OS_SYSTEM                          Linux
ROOT_URL                           https://{SERVER_NAME}
SCM_ID                             v2017.07.13-103-gfac514b
SERVER_NAME                        localhost
TOP_DIR                            /home/mrohan/clients/osstp-yc
USER                               mrohan
VIRTUAL_ENV                        /home/mrohan/clients/venv
WEBAPPS_DIR                        {BASE_DIR}
YAMLCONF_SYSFILES_DIR              {BASE_DIR}/osstpmgt/templates/sys

Use "ycexplain" for more information on individual attributes
```

2.1.3 `ycsysfiles` Command

The `ycsysfiles` management command supports the creation of system control files, e.g., Apache configuration files, based on the attributes defined via YAMLCONF files. The command scans the directory defined by by the attribute, e.g.,:

```
YAMLCONF_SYSFILES_DIR: '{BASE_DIR}/templates/sys'
```

for each file found, it

1. Maps it to a file system path by stripping the `YAMLCONF_SYSFILES_DIR` prefix and expanding attribute references (yes, that paths under this directory will contain `{` and `}` characters).
2. If the mapped file exists and is writable to the user running the `ycsysfiles` command, it is updated with the contents generated by Django template engine with YAMLCONF defined attributes being available for substitution in the templates or use for conditionals.

For example, the Django tutorial implementations under the `examples` directory contains, within the `mysite/templates/sys` directory, the template files:

1. `etc/apache2/sites-available/mysite.conf`, this template would be used to create the system file `/etc/apache2/sites-available/mysite.conf` (the Apache site config file on an Ubuntu system).
2. `{BASE_DIR}/sysfiles.txt`, this template would be used to create the file `sysfiles.txt` relative the directory where the Django application is installed. E.g., if installed in `/var/mysite`, the file `/var/mysite/sysfiles.txt` would be created.

The paths under the `YAMLCONF_SYSFILES_DIR` directory can reference YAMLCONF defined attributes via standard Python key based format references, as with `BASE_DIR` above.

The attributes available can be extended using the `YAMLCONF_ATTRIBUTE_FUNCTIONS` attribtue. This makes attributes based on, e.g., the contents of the Django application database available when processing files. A contrived example would be, in a `ycattrs.py` file (conventionally in the same directory as the `settings.py` file):

```
def userlist():
    return {
        'USERS': User.objects.all(),
    }
```

2.2 Support for Dictionaries

YAMLCONF uses the "." character to identify attributes defined as part of a dictionary, e.g., the DATABASES attribute. To set, e.g., the password for a database connection:

```
DATABASES.default.PASSWORD: some-secret-password
```

It is considered an error if dotted name refers to a settings attribute that is not an dictionary, the setting is ignored by YAMLCONF.

The dotted notation should be used to update dictionaries already defined in the settings file. To add a new dictionary, a YAML dictionary definition should be used, e.g.,:

```
NEW_DICTIONARY:
    key1: value1
    key2: value2
```

2.3 Attribute Substitution

Frequently, attributes values are defined in terms of other attribute values, most commonly using the base directory to define other directories. The YAMLCONF allows other attributes to be referenced using the Python named formatting syntax, e.g.,:

```
LOG_DIR: "{BASE_DIR}/log"
```

Currently only attributes defined via YAML files can be used in this way. To disable this on a per-attribute basis, the `:raw` qualifier should be defined to modify the behaviour for attribute, e.g.,:

```
LOGGING.formatters.simple.format: '%(asctime)s %(levelname)s %(message)s'
LOGGING.formatters.simple.format:raw: True
```

2.4 Hiding values

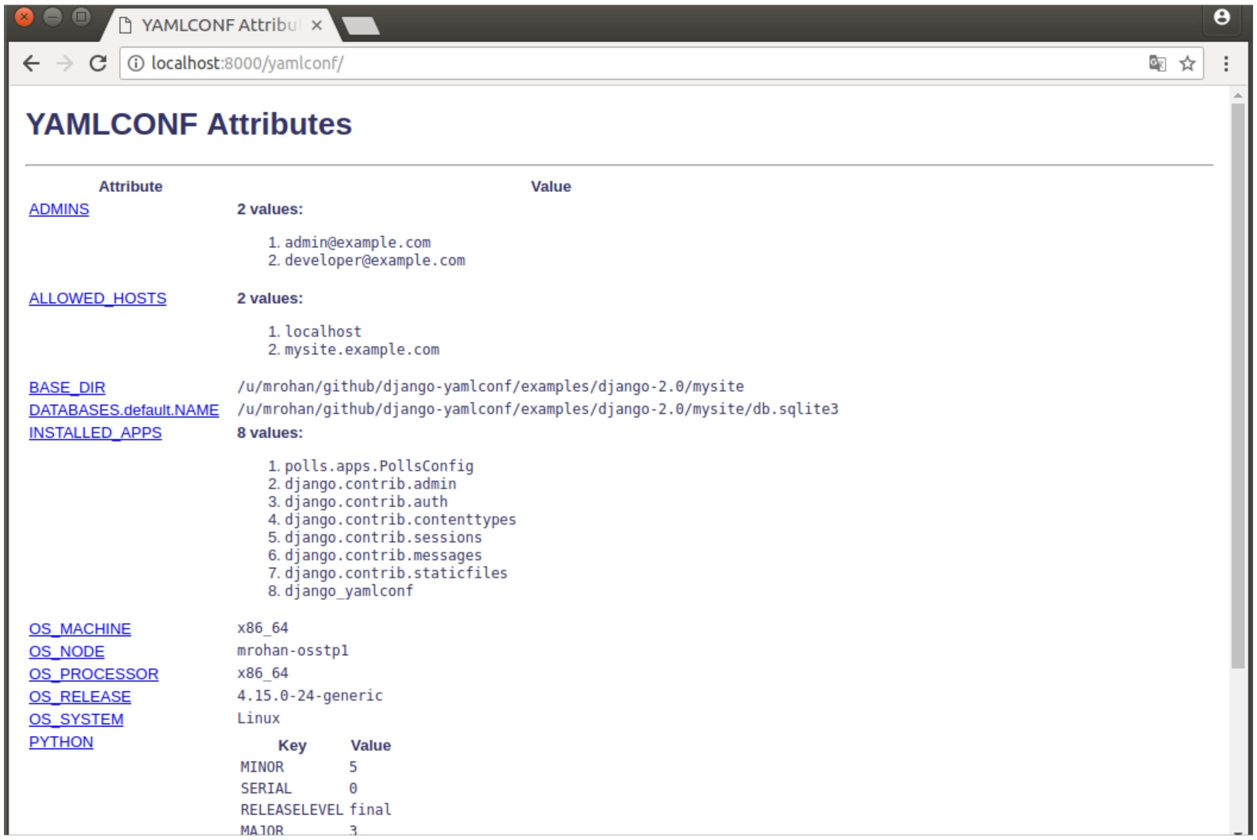
The YAMLCONF includes an experimental view to handle URLs to display attributes (should only be used in a debugging context), e.g., adding the URL definition to your application:

```
url(r'^yamlconf/', include('django_yamlconf.urls')),
```

will display the YAMLCONF attributes. For older versions of Django, the `namespace` needs to be explictly defined:

```
url(r'^yamlconf/', include('django_yamlconf.urls', namespace='django_yamlconf')),
```

An example of the page displayed is:



YAMLCONF Attributes

Attribute	Value
ADMINS	**2 values:** 1. `admin@example.com` 2. `developer@example.com`
ALLOWED_HOSTS	**2 values:** 1. `localhost` 2. `mysite.example.com`
BASE_DIR	`/u/mrohan/github/django-yamlconf/examples/django-2.0/mysite`
DATABASES.default.NAME	`/u/mrohan/github/django-yamlconf/examples/django-2.0/mysite/db.sqlite3`
INSTALLED_APPS	**8 values:** 1. `polls.apps.PollsConfig` 2. `django.contrib.admin` 3. `django.contrib.auth` 4. `django.contrib.contenttypes` 5. `django.contrib.sessions` 6. `django.contrib.messages` 7. `django.contrib.staticfiles` 8. `django_yamlconf`
OS_MACHINE	`x86_64`
OS_NODE	`mrohan-osstp1`
OS_PROCESSOR	`x86_64`
OS_RELEASE	`4.15.0-24-generic`
OS_SYSTEM	`Linux`

PYTHON	**Key** / **Value**
	`MINOR` `5`
	`SERIAL` `0`
	`RELEASELEVEL` `final`
	`MAJOR` `3`

Fig. 1: Attributes Index Page

By default, any attribute value with the string `PASSWORD` in the name will have their values hidden in the HTML displayed. Other, sensitive, values can be explicitly hidden by defining the qualifier attribute `:hide`, e.g.,:

```
APIKEY: 'my-api-key'
APIKEY:hide: True
```

2.5 Extending Values

For list values, the qualifier attributes `:prepend` and `:append` can be used to extend the underlying definition, e.g., add another admin user, the following definition can be used:

```
ADMINS:append: 'someuser@vmware.com'
```

The value of `:prepend` or `:append` qualified attribute can be either a single value, as above, or a list of values. When a list is given, the attribute is extend with the extra values, e.g.,:

```
ADMINS:append:
  - 'someuser1@vmware.com'
  - 'someuser2@vmware.com'
```

Normally, list values in the settings file are simply unordered lists. There are, however, some values where the order matters, in particular, the `MIDDLEWARE` list. A middleware that short-circuits the handling of requests would need to be placed at the beginning of the list. This is the rationale for the `:prepend` functionality.

2.6 Pre-defined Attributes

The YAMLCONF module predefines the following attributes which can be used, along with other attributed defined, via attribute substitution:

`BASE_DIR` The directory containing the `setting.py` file

`PYTHON` This is a dictionary giving the major, minor, micro, releaselevel serial values for the Python interpretor

`OS_MACHINE` The value of the `platform.machine()` function, e.g., `x86_64`

`OS_NODE` The value of the `platform.node()` function, the system short name

`OS_PROCESSOR` The value of the `platform.machine()` function, e.g., `x86_64`

`OS_RELEASE` The value of the `platform.release()` function, e.g., `4.4.0-101-generic`

`OS_SYSTEM` The value of the `platform.system()` function, e.g., `Linux`

`TOP_DIR` The directory above BASE_DIR

`USER` The login name of the current user

`VIRTUAL_ENV` If run within a Python virtual environment, this attribute is defined to be the path to the environment, otherwise it has the value `None`

2.7 Attribute Documentation

Appending `:doc` to an attribute name in a YAML file defines a documentation string for the attribute. This should be used to give information on the expected value for the attribute and how the value might differ on production, beta and development servers, e.g., documentation for the DEBUG attribute would be defined using the YAML:

```
DEBUG:doc: |
    Enable or disable debugging functionality.  On the production server
    this attribute should be set to false
```

2.8 Typical Structure

On a typical production system for the "buildaudit" app, a local `buildaudit.yaml` would exist in, e.g., the `/var/www` directory. This would contain the production passwords, debug settings, etc. Under this directory, a `webapps` directory could contain another `buildaudit.yaml` file possibly generated by a build process which could define attributes identifying the build, the Git Hash for the code, build time, etc. Finally, a `buildaudit.yaml` file co-located with the settings.py file giving the base attributes and their documentation strings:

```
+- /var/www
    +- buildaudit.yaml
    +- webapps
       +- buildaudit.yaml
       +- buildaudit
           +- buildaudit.yaml
           +- settings.py
```

2.9 Environment Variables

As a final source for values, the environment is queries for all environment names beginning with `YAMLCONF_`. E.g., to "inject" the value "xyx" for the setting "XYZ", the environment can be used:

```
$ export YAMLCONF_XYZ=xyz
```

Environment variable values are pulled into the settings as a simple string value. For more complex values, the environment value can be interpreted as a JSON encode structure if a setting with the `:jsonenv` qualifier is True for the setting. E.g., in a Fabric base deployment system, the servers to deploy to can be defined in the base YAMLCONF file as:

```
DEPLOY_SERVERS:
  - '{DEPLOY_USER}@localhost'
DEPLOY_USER: '{USER}'
```

I.e., deploy to `localhost` as the current user. In a production environment, the production servers would likely be a list of servers behind an HA-Proxy server. This list can be defined via a local YAMLCONF file in the directory tree on the system where deployments are run. A local file can, however, be awkward in some contexts, e.g., deploy occurs as a Concourse job, and an environment variable definition is easier. In this case, the value can be a JSON encoded value and JSON decode enabled via the `:jsonenv` qualifier. The base YAMLCONF file would now include the definitions:

```
DEPLOY_SERVERS:
  - '{DEPLOY_USER}@localhost'
DEPLOY_SERVERS:jsonenv: True
```

and the list of servers to deploy to "injected" via an environment variable:

```
$ export YAMLCONF_DEPLOY_SERVERS='["{DEPLOY_USER}@host-a", "{DEPLOY_USER}@host-b"]'
```

2.10 Public Methods

The primary public method is the `load` method which loads the attribute definitions from YAML file located in the directory tree. Other methods are exported, and are documented here, but it is expected that these methods are only used by the management commands.

2.10.1 `add_attributes` Function

`django_yamlconf.`**`add_attributes`**(*settings*, *attributes*, *source*)

Add a set of name value pairs to the set of attributes, e.g., attributes defined on the command line for management commands. Since this occurs after Django has loaded the settings, this function *does not*, in general, change behaviour of Django. It is used to add attribute definitions from management command lines. While this does not impact the behaviour of Django, it does make the attributes available for use in templates for the `ycsysfiles` command.

Parameters

- **`settings`** – the Django settings module
- **`attributes`** – the dictionary of name/values pairs to add
- **`source`** – the name for the source (displayed by `ycexplain`)

Returns *None*

2.10.2 `defined_attributes` Function

`django_yamlconf.`**`defined_attributes`**(*settings=None*, *template_use=False*)

Return a dictionary giving attribute names and associated values. This dictionary can be used as the variables when rendering templates. This is the set attributed used used as the variables when rendering templates for the `ycsysfiles` command.

Parameters

- **`settings`** – the Django settings module (this is optional, defaults to the settings modules used when loading)
- **`template_use`** – If the the set of attributes return needs to be used to process a template, in the dictionary returned, attribute keys are added for dictionary parents e.g., “DATABASES”, if “DATABASES.default. . . ” is a YAMLCONF defined attribute. The usage without this option support the *yclist* management command.

Returns a dictionary giving attribute names and associated values.

Return type dict

2.10.3 `explain` Function

`django_yamlconf.`**`explain`**(*name*, *settings=None*, *stream=None*)

Explain the source for an attribute definition including sources that were eclipsed by higher level YAML definition files. If the attribute has associated documentation, it is also printed.

This routine is only used by the YAMLCONF management command `ycexplain`.

Parameters

- **`name`** – the YAMLCONF controlled setting name

- **settings** – the Django settings module
- **stream** – the stream to write the explanation text (defaults to `sys.stdout`)

Returns *None*

2.10.4 `list_attrs` Function

`django_yamlconf.list_attrs`(*settings=None*, *stream=None*)

Write a list of attributes managed by YAMLCONF to the given stream (defaults to `sys.stdout`). Additional information can be printed using the `explain` routine.

This routine is only used by the YAMLCONF management command `yclist`.

Parameters

- **settings** – the Django settings module
- **stream** – the stream to write the list text

Returns *None*

2.10.5 `load` Function

`django_yamlconf.load`(*syntax='yaml'*, *settings=None*, *base_dir=None*, *project=None*)

Load the set of YAML files for a Django project. The simplest usage is to call this at the end of a settings file. In this context, no arguments are needed.

Parameters

- **syntax** – The "syntax" parameter should name a Python module with a "load" method, e.g., the default is "yaml.load". Other possibiliities could be "json" to use JSON formatted file or, even, "pickle" but that would be strange. The "syntax" name is also used as the file extension for the YAMLCONF files.
- **settings** – The "settings" should be module containing the Django settings. This is determined from the call stack if no module is given.
- **base_dir** – The "base_dir" defines the starting directory for YAMLCONF files and defaults to the directory containing the settings module.
- **project** – The "project" is the name of the Django project and defaults to the name of the directory containing the settings modules.

Returns *None*

2.10.6 `sysfiles` Function

`django_yamlconf.sysfiles`(*create*, *noop*, *settings*, *rootdir=''*, *render=None*)

Traverse the sys templates directory expanding files to the destination directory.

Parameters

- **create** – the template files should be created, normally will only update files that already exist on the system and are writable.
- **noop** – no-op mode, print what would be done.
- **settings** – the Django settings module

- **rootdir** – the directory to create the system files, defaults to /, i.e., the root file system.
- **render** – the rendering engine, if not given, defaults to Django's render_to_string

Returns *None*

2.11 Examples

The examples are based on the polls example from the Django Project web site. There are two flavors of this example:

1. Under Django version 2.0 in the directory examples/django-2.0
2. Under Django version 3.0 in the directory examples/django-3.0

The django-1.11 directory has been removed as it is end of life and GitHub is generating secuity issues on the old dependencies.

See the Examples Directory on GitHub.

The examples for both versions of Django behaviour similarly: there are Makefile targets to:

- init initialize a local SQLite database for the application (should be the first target executed, if experimenting.
- runserver to run a local server
- General utility targets for YAMLCONF: yclist, ycexplain and ycsysfiles.

An example of the usage of YAMLCONF, would be, e.g., in a production environment, switching to a PostgreSQL database via the creation of a mysite.yaml file (would need to explicitly install the psycopg2-binary module):

```
DATABASES.default:
    ENGINE: django.db.backends.postgresql_psycopg2
    NAME: mysite
    USER: mysite
    PASSWORD: my-password
    HOST: localhost
    PORT: ''
```

2.12 Limitations

Some of the current limitations for this implementation are:

- Currently cannot substitute list values, e.g.,:

```
ADMINS:
  - jsmith
  - auser
MANAGER: "{ADMINS}"
```

- The pre-defined attributes should also include the host IP address

These might be addressed if the need arises.

2.13 Releases & Major Branches

2.13.1 Version 1.4.0

- Tagged with `v1.4.0`.
- Added support for JSON encoded environment values if decorated with `:jsonenv`. If JSON decoding fails (invalid JSON string), the value is used as is. This allows the definition of more complex values via the environment, list, dictionaries, etc. This can be used in K8s environments, e.g., Concourse
- Added a `CODE-OF-CONDUCT` file for contributors.

2.13.2 Version 1.3.0

- Tagged with `v1.3.0`.
- Dropped explicit dependency on Django for package. Overall project should include Django but also allows usage of package outside of a Django project.
- Update to support Django 3.0 (staticfiles -> static in template)
- Added Django 3.0 example (polls)
- Removed the Django 1.x example (polls)

2.13.3 Version 1.2.1

- Tagged with `v1.2.1`.
- Fix generation of the long description for the package.

2.13.4 Version 1.2.0

- Tagged with `v1.2.0`.
- Updates to support Django 3.0: Simply use "*six*" instead of the support "*django.utils.six*" package and use "*render*" instead of "*render_to_response*".
- *ycsysfiles* should generate executable files if the source template file is executable.
- Ensure the absolute path is used when searching for YAML control files. This issue is seen when running Django apps under uWSGI control.
- Added the built-in attribute `CPU_COUNT` (primarily for use in uWSGI ini files) giving the number of available CPUs.

2.13.5 Version 1.1.0

- Tagged with `v1.1.0`.
- Handle stricter loading for newer versions of PyYAML. The warning "YAMLLoadWarning: calling yaml.load() without Loader=... is deprecated" is generated referring to https://msg.pyyaml.org/load for full details. The YAML load now specified Loader=FullLoader.

- The `defined_attributes` function now returns a dictionary with additional keys if the attribute defined is a nested dictionary, the top level dictionary from the setting file is now also added, e.g., if "`DATABASES.default`" is defined, the value returned will now also have a "`DATABASES`" key.
- Added `docs` directory and Sphinx infrastructure to support publishing to readthedocs.org
- Added support for a final, environment defined, YAML file defined via the environment variable *YAMLCONF_CONFFILE*

2.13.6 Version 1.0.0

- Initial public release (tagged with `v1.0.0`)

2.14 Contributing

The `django-yamlconf` project team welcomes contributions from the community. Before you start working with `django-yamlconf`, please read our Developer Certificate of Origin. All contributions to this repository must be signed as described on that page. Your signature certifies that you wrote the patch or have the right to pass it on as an open-source patch. For more detailed information, refer to CONTRIBUTING.md.

2.15 Authors

Created and maintained by Michael Rohan mrohan@vmware.com

Index