django-configurations Documentation

Release dev

Jannis Leidel

Jul 16, 2017

Contents

1 Quickstart **3**
- 1.1 Project templates . . . 4

2 Wait, what? **5**

3 Okay, how does it work? **7**

4 But isn't that magic? **9**

5 Further documentation **11**
- 5.1 Usage patterns . . . 11
  - 5.1.1 Server specific settings . . . 11
  - 5.1.2 Global settings defaults . . . 12
  - 5.1.3 Configuration mixins . . . 12
  - 5.1.4 Pristine methods . . . 12
  - 5.1.5 Setup methods . . . 13
  - 5.1.6 Standalone scripts . . . 14
- 5.2 Values . . . 14
  - 5.2.1 Overview . . . 14
  - 5.2.2 Environment variables . . . 15
  - 5.2.3 `Value` class . . . 16
  - 5.2.4 Built-ins . . . 17
- 5.3 Cookbook . . . 23
  - 5.3.1 Calling a Django management command . . . 23
  - 5.3.2 Envdir . . . 23
  - 5.3.3 Project templates . . . 23
  - 5.3.4 Celery . . . 24
  - 5.3.5 iPython notebooks . . . 25
  - 5.3.6 FastCGI . . . 25
  - 5.3.7 Sphinx . . . 25
- 5.4 Changelog . . . 26
  - 5.4.1 v2.0 (2016-07-29) . . . 26
  - 5.4.2 v1.0 (2016-01-04) . . . 26
  - 5.4.3 v0.8 (2014-01-16) . . . 27
  - 5.4.4 v0.7 (2013-11-26) . . . 27
  - 5.4.5 v0.6 (2013-09-19) . . . 27
  - 5.4.6 v0.5.1 (2013-09-12) . . . 27

5.4.7 v0.5 (2013-09-09) . . . 27
5.4.8 v0.4 (2013-09-03) . . . 27
5.4.9 v0.3.2 (2014-01-16) . . . 28
5.4.10 v0.3.1 (2013-09-20) . . . 28
5.4.11 v0.3 (2013-05-15) . . . 28
5.4.12 v0.2.1 (2013-04-11) . . . 28
5.4.13 v0.2 (2013-03-27) . . . 28
5.4.14 v0.1 (2012-07-21) . . . 28

6 Alternatives **29**

7 Bugs and feature requests **31**

Python Module Index **33**

django-configurations eases Django project configuration by relying on the composability of Python classes. It extends the notion of Django's module based settings loading with well established object oriented programming patterns.

Check out the documentation for more complete examples.

CHAPTER 1

Quickstart

Install django-configurations:

```
pip install django-configurations
```

Then subclass the included `configurations.Configuration` class in your project's **settings.py** or any other module you're using to store the settings constants, e.g.:

```
# mysite/settings.py

from configurations import Configuration

class Dev(Configuration):
    DEBUG = True
```

Set the `DJANGO_CONFIGURATION` environment variable to the name of the class you just created, e.g. in bash:

```
export DJANGO_CONFIGURATION=Dev
```

and the `DJANGO_SETTINGS_MODULE` environment variable to the module import path as usual, e.g. in bash:

```
export DJANGO_SETTINGS_MODULE=mysite.settings
```

Alternatively supply the `--configuration` option when using Django management commands along the lines of Django's default `--settings` command line option, e.g.:

```
python manage.py runserver --settings=mysite.settings --configuration=Dev
```

To enable Django to use your configuration you now have to modify your **manage.py** or **wsgi.py** script to use django-configurations's versions of the appropriate starter functions, e.g. a typical **manage.py** using django-configurations would look like this:

```
#!/usr/bin/env python

import os
```

```
import sys

if __name__ == "__main__":
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
    os.environ.setdefault('DJANGO_CONFIGURATION', 'Dev')

    from configurations.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

Notice in line 10 we don't use the common tool `django.core.management.execute_from_command_line` but instead `configurations.management.execute_from_command_line`.

The same applies to your **wsgi.py** file, e.g.:

```
import os

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
os.environ.setdefault('DJANGO_CONFIGURATION', 'Dev')

from configurations.wsgi import get_wsgi_application

application = get_wsgi_application()
```

Here we don't use the default `django.core.wsgi.get_wsgi_application` function but instead `configurations.wsgi.get_wsgi_application`.

That's it! You can now use your project with `manage.py` and your favorite WSGI enabled server.

Project templates

Don't miss the Django *project templates pre-configured with django-configurations* to simplify getting started with new Django projects.

CHAPTER 2

Wait, what?

django-configurations helps you organize the configuration of your Django project by providing the glue code to bridge between Django's module based settings system and programming patterns like mixins, facades, factories and adapters that are useful for non-trivial configuration scenarios.

It allows you to use the native abilities of Python inheritance without the side effects of module level namespaces that often lead to the unfortunate use of the `from foo import *` anti-pattern.

CHAPTER 3

Okay, how does it work?

Any subclass of the `configurations.Configuration` class will automatically use the values of its class and instance attributes (including properties and methods) to set module level variables of the same module – that's how Django will interface to the django-configurations based settings during startup and also the reason why it requires you to use its own startup functions.

That means when Django starts up django-configurations will have a look at the `DJANGO_CONFIGURATION` environment variable to figure out which class in the settings module (as defined by the `DJANGO_SETTINGS_MODULE` environment variable) should be used for the process. It then instantiates the class defined with `DJANGO_CONFIGURATION` and copies the uppercase attributes to the module level variables.

New in version 0.2.

Alternatively you can use the `--configuration` command line option that django-configurations adds to all Django management commands. Behind the scenes it will simply set the `DJANGO_CONFIGURATION` environement variable so this is purely optional and just there to compliment the default `--settings` option that Django adds if you prefer that instead of setting environment variables.

CHAPTER 4

But isn't that magic?

Yes, it looks like magic, but it's also maintainable and non-intrusive. No monkey patching is needed to teach Django how to load settings via django-configurations because it uses Python import hooks (PEP 302) behind the scenes.

CHAPTER 5

Further documentation

Usage patterns

There are various configuration patterns that can be implemented with django-configurations. The most common pattern is to have a base class and various subclasses based on the enviroment they are supposed to be used in, e.g. in production, staging and development.

Server specific settings

For example, imagine you have a base setting class in your **settings.py** file:

```
from configurations import Configuration

class Base(Configuration):
    TIME_ZONE = 'Europe/Berlin'

class Dev(Base):
    DEBUG = True
    TEMPLATE_DEBUG = DEBUG

class Prod(Base):
    TIME_ZONE = 'America/New_York'
```

You can now set the `DJANGO_CONFIGURATION` environment variable to one of the class names you've defined, e.g. on your production server it should be `Prod`. In bash that would be:

```
export DJANGO_SETTINGS_MODULE=mysite.settings
export DJANGO_CONFIGURATION=Prod
python manage.py runserver
```

Alternatively you can use the `--configuration` option when using Django management commands along the lines of Django's default `--settings` command line option, e.g.:

```
python manage.py runserver --settings=mysite.settings --configuration=Prod
```

Global settings defaults

Every `configurations.Configuration` subclass will automatically contain Django's global settings as class attributes, so you can refer to them when setting other values, e.g.:

```
from configurations import Configuration

class Prod(Configuration):
    TEMPLATE_CONTEXT_PROCESSORS = Configuration.TEMPLATE_CONTEXT_PROCESSORS + (
            'django.core.context_processors.request',
        )

    @property
    def LANGUAGES(self):
        return list(Configuration.LANGUAGES) + [('tlh', 'Klingon')]
```

Configuration mixins

You might want to apply some configuration values for each and every project you're working on without having to repeat yourself. Just define a few mixin you re-use multiple times:

```
class FullPageCaching(object):
    USE_ETAGS = True
```

Then import that mixin class in your site settings module and use it with a `Configuration` class:

```
from configurations import Configuration

class Prod(FullPageCaching, Configuration):
    DEBUG = False
    # ...
```

Pristine methods

New in version 0.3.

In case one of your settings itself need to be a callable, you need to tell that django-configurations by using the `pristinemethod` decorator, e.g.:

```
from configurations import Configuration, pristinemethod

class Prod(Configuration):

    @pristinemethod
    def ACCESS_FUNCTION(user):
        return user.is_staff
```

Lambdas work, too:

```
from configurations import Configuration, pristinemethod

class Prod(Configuration):
    ACCESS_FUNCTION = pristinemethod(lambda user: user.is_staff)
```

Setup methods

New in version 0.3.

If there is something required to be set up before, during or after the settings loading happens, please override the `pre_setup`, `setup` or `post_setup` class methods like so (don't forget to apply the Python `@classmethod` decorator):

```
import logging
from configurations import Configuration

class Prod(Configuration):
    # ...

    @classmethod
    def pre_setup(cls):
        super(Prod, cls).pre_setup()
        if something.completely.different():
            cls.DEBUG = True

    @classmethod
    def setup(cls):
        super(Prod, cls).setup()
        logging.info('production settings loaded: %s', cls)

    @classmethod
    def post_setup(cls):
        super(Prod, cls).post_setup()
        logging.debug("done setting up! \o/")
```

As you can see above the `pre_setup` method can also be used to programmatically change a class attribute of the settings class and it will be taken into account when doing the rest of the settings setup. Of course that won't work for `post_setup` since that's when the settings setup is already done.

In fact you can easily do something unrelated to settings, like connecting to a database:

```
from configurations import Configuration

class Prod(Configuration):
    # ...

    @classmethod
    def post_setup(cls):
        import mango
        mango.connect('enterprise')
```

Warning: You could do the same by overriding the `__init__` method of your settings class but this may cause hard to debug errors because at the time the `__init__` method is called (during Django startup) the Django setting system isn't fully loaded yet.

> So anything you do in `__init__` that may require `django.conf.settings` or Django models there is a good chance it won't work. Use the `post_setup` method for that instead.

Changed in version 0.4: A new `setup` method was added to be able to handle the new *`Value`* classes and allow an in-between modification of the configuration values.

Standalone scripts

If you want to run scripts outside of your project you need to add these lines on top of your file:

```
import configurations
configurations.setup()
```

Values

New in version 0.4.

django-configurations allows you to optionally reduce the amount of validation and setup code in your **settings.py** by using `Value` classes. They have the ability to handle values from the process environment of your software (`os.environ`) and work well in projects that follow the Twelve-Factor methodology.

Note: These classes are required to be used as attributes of `Configuration` classes. See the *main documentation* for more information.

Overview

Here is an example (from a **settings.py** file with a `Configuration` subclass):

```
from configurations import Configuration, values

class Dev(Configuration):
    DEBUG = values.BooleanValue(True)
```

As you can see all you have to do is to wrap your settings value in a call to one of the included values classes. When Django's process starts up it will automatically make sure the passed-in value validates correctly – in the above case checks if the value is really a boolean.

You can safely use other *`Value`* instances as the default setting value:

```
from configurations import Configuration, values

class Dev(Configuration):
    DEBUG = values.BooleanValue(True)
    TEMPLATE_DEBUG = values.BooleanValue(DEBUG)
```

See the list of *built-in value classes* for more information.

Environment variables

To separate the site configuration from your application code you should use environment variables for configuration. Unfortunately environment variables are string based so they are not easily mapped to the Python based settings system Django uses.

Luckily django-configurations' *Value* subclasses have the ability to handle environment variables for the common use cases.

Default behavior

For example, imagine you want to override the ROOT_URLCONF setting on your staging server to be able to debug a problem with your in-development code. You're using a web server that passes the environment variables from the shell it was started from into your Django WSGI process.

Use the boolean environ option of the *Value* class (True by default) to tell django-configurations to look for an environment variable with the same name as the specific *Value* variable, only uppercased and prefixed with DJANGO_. E.g.:

```
from configurations import Configuration, values

class Stage(Configuration):
    # ..
    ROOT_URLCONF = values.Value('mysite.urls')
```

django-configurations will try to read the DJANGO_ROOT_URLCONF environment variable when deciding which value the ROOT_URLCONF setting should have. When you run the web server simply specify that environment variable (e.g. in your init script):

```
DJANGO_ROOT_URLCONF=mysite.debugging_urls gunicorn mysite.wsgi:application
```

If the environment variable can't be found it'll use the default 'mysite.urls'.

Disabling environment variables

To disable environment variables, specify the environ parameter of the *Value* class. For example this would disable it for the TIME_ZONE setting value:

```
from configurations import Configuration, values

class Dev(Configuration):
    TIME_ZONE = values.Value('UTC', environ=False)
```

Custom environment variable names

To support legacy systems, integrate with other parts of your software stack or simply better match your taste in naming public configuration variables, django-configurations allows you to use the environ_name parameter of the *Value* class to change the base name of the environment variable it looks for. For example this would enforce the name DJANGO_MYSITE_TZ instead of using the name of the *Value* instance.:

```
from configurations import Configuration, values

class Dev(Configuration):
    TIME_ZONE = values.Value('UTC', environ_name='MYSITE_TZ')
```

Allow final value to be used outside the configuration context

You may use the `environ_name` parameter to allow a *`Value`* to be directly converted to its final value for use outside of the configuration context:

```
>>> type(values.Value([]))
<class 'configurations.values.Value'>
>>> type(values.Value([], environ_name="FOOBAR"))
<class 'list'>
```

This can also be achieved when using `environ=False` and providing a default value.

Custom environment variable prefixes

In case you want to change the default environment variable name prefix of `DJANGO` to something to your likening, use the `environ_prefix` parameter of the *`Value`* instance. Here it'll look for the `MYSITE_TIME_ZONE` environment variable (instead of `DJANGO_TIME_ZONE`):

```
from configurations import Configuration, values

class Dev(Configuration):
    TIME_ZONE = values.Value('UTC', environ_prefix='MYSITE')
```

The `environ_prefix` parameter can also be `None` to completely disable the prefix.

`Value` class

class `Value` (*default*[, *environ=True*, *environ_name=None*, *environ_prefix='DJANGO'*, *environ_required=False*])

The `Value` class takes one required and several optional parameters.

Parameters

- **`default`** – the default value of the setting
- **`environ`** (*`bool`*) – toggle for environment use
- **`environ_name`** (*`capitalized string or None`*) – name of environment variable to look for
- **`environ_prefix`** (*`capitalized string`*) – prefix to use when looking for environment variable
- **`environ_required`** (*`bool`*) – wheter or not the value is required to be set as an environment variable

The `default` parameter is effectively the value the setting has right now in your `settings.py`.

`setup` (*name*)

Parameters **`name`** – the name of the setting

Returns setting value

The `setup` method is called during startup of the Django process and implements the ability to check the environment variable. Its purpose is to return a value django-configurations is supposed to use when loading the settings. It'll be passed one parameter, the name of the *`Value`* instance as defined in the `settings.py`. This is used for building the name of the environment variable.

to_python(*value*)

Parameters **value** – the value of the setting as found in the process environment (`os.environ`)

Returns validated and "ready" setting value if found in process environment

The `to_python` method is used when the `environ` parameter of the *Value* class is set to `True` (the default) and an environment variable with the appropriate name was found.

It will be used to handle the string based environment variables and returns the "ready" value of the setting.

Some *Value* subclasses also use it during initialization when the default value has a string-like format like an environment variable which needs to be converted into a Python data type.

Built-ins

Type values

class BooleanValue

A *Value* subclass that checks and returns boolean values. Possible values for environment variables are:

- `True` values: `'yes'`, `'y'`, `'true'`, `'1'`
- `False` values: `'no'`, `'n'`, `'false'`, `'0'`, `''` (empty string)

```
DEBUG = values.BooleanValue(True)
```

class IntegerValue

A *Value* subclass that handles integer values.

```
MYSITE_CACHE_TIMEOUT = values.IntegerValue(3600)
```

class FloatValue

A *Value* subclass that handles float values.

```
MYSITE_TAX_RATE = values.FloatValue(11.9)
```

class DecimalValue

A *Value* subclass that handles Decimal values.

```
MYSITE_CONVERSION_RATE = values.DecimalValue(decimal.Decimal('4.56214'))
```

class ListValue(*default*[, *separator=', ', converter=None*])

A `SequenceValue` subclass that handles list values.

Parameters

- **separator** – the separator to split environment variables with
- **converter** – the optional converter callable to apply for each list item

Simple example:

```
ALLOWED_HOSTS = ListValue(['mysite.com', 'mysite.biz'])
```

Use a custom converter to check for the given variables:

```
def check_monty_python(person):
    if not is_completely_different(person):
        error = '{0} is not a Monty Python member'.format(person)
        raise ValueError(error)
    return person

MONTY_PYTHONS = ListValue(['John Cleese', 'Eric Idle'],
                          converter=check_monty_python)
```

You can override this list with an environment variable like this:

```
DJANGO_MONTY_PYTHONS="Terry Jones,Graham Chapman" gunicorn mysite.wsgi:application
```

Use a custom separator:

```
EMERGENCY_EMAILS = ListValue(['admin@mysite.net'], separator=';')
```

And override it:

```
DJANGO_EMERGENCY_EMAILS="admin@mysite.net;manager@mysite.org;support@mysite.com"␣
↪gunicorn mysite.wsgi:application
```

class `TupleValue`

A `SequenceValue` subclass that handles tuple values.

Parameters

- **`separator`** – the separator to split environment variables with
- **`converter`** – the optional converter callable to apply for each tuple item

See the *`ListValue`* examples above.

class `SingleNestedTupleValue`(*default*[, *seq_separator=';'*, *separator=' '*, *converter=None*])

A `SingleNestedSequenceValue` subclass that handles single nested tuple values, e.g. `((a, b), (c, d))`.

Parameters

- **`seq_separator`** – the separator to split each tuple with
- **`separator`** – the separator to split the inner tuple contents with
- **`converter`** – the optional converter callable to apply for each inner tuple item

Useful for ADMINS, MANAGERS, and the like. For example:

```
ADMINS = SingleNestedTupleValue((
    ('John', 'jcleese@site.com'),
    ('Eric', 'eidle@site.com'),
))
```

Override using environment variables like this:

```
DJANGO_ADMINS=Terry,tjones@site.com;Graham,gchapman@site.com
```

class `SingleNestedListValue`(*default*[, *seq_separator=';'*, *separator=' '*, *converter=None*])

A `SingleNestedSequenceValue` subclass that handles single nested list values, e.g. `[[a, b], [c, d]]`.

Parameters

- **seq_separator** – the separator to split each list with
- **separator** – the separator to split the inner list contents with
- **converter** – the optional converter callable to apply for each inner list item

See the *SingleNestedTupleValue* examples above.

class SetValue

A *Value* subclass that handles set values.

Parameters

- **separator** – the separator to split environment variables with
- **converter** – the optional converter callable to apply for each set item

See the *ListValue* examples above.

class DictValue

A *Value* subclass that handles dicts.

```
DEPARTMENTS = values.DictValue({
    'it': ['Mike', 'Joe'],
})
```

Validator values

class EmailValue

A *Value* subclass that validates the value using the `django.core.validators.validate_email` validator.

```
SUPPORT_EMAIL = values.EmailValue('support@mysite.com')
```

class URLValue

A *Value* subclass that validates the value using the `django.core.validators.URLValidator` validator.

```
SUPPORT_URL = values.URLValue('https://support.mysite.com/')
```

class IPValue

A *Value* subclass that validates the value using the `django.core.validators.validate_ipv46_address` validator.

```
LOADBALANCER_IP = values.IPValue('127.0.0.1')
```

class RegexValue (*default*, *regex*[, *environ=True*, *environ_name=None*, *environ_prefix='DJANGO'*])

A *Value* subclass that validates according a regular expression and uses the `django.core.validators.RegexValidator`.

Parameters **regex** – the regular expression

```
DEFAULT_SKU = values.RegexValue('000-000-00', regex=r'\d{3}-\d{3}-\d{2}')
```

class PathValue (*default*[, *check_exists=True*, *environ=True*, *environ_name=None*, *environ_prefix='DJANGO'*])

A *Value* subclass that normalizes the given path using `os.path.expanduser` and checks if it exists on the file system.

Takes an optional `check_exists` parameter to disable the check with `os.path.exists`.

Parameters **check_exists** – toggle the file system check

```
BASE_DIR = values.PathValue('/opt/mysite/')
STATIC_ROOT = values.PathValue('/var/www/static', checks_exists=False)
```

URL-based values

Note: The following URL-based *Value* subclasses are inspired by the Twelve-Factor methodology and use environment variable names that are already established by that methodology, e.g. `'DATABASE_URL'`.

Each of these classes require external libraries to be installed, e.g. the *DatabaseURLValue* class depends on the package `dj-database-url`. See the specific class documentation below for which package is needed.

---

class **DatabaseURLValue**(*default*[, *alias='default'*, *environ=True*, *environ_name='DATABASE_URL'*, *environ_prefix=None*])

A *Value* subclass that uses the dj-database-url app to convert a database configuration value stored in the `DATABASE_URL` environment variable into an appropriate setting value. It's inspired by the Twelve-Factor methodology.

By default this *Value* subclass looks for the `DATABASE_URL` environment variable.

Takes an optional `alias` parameter to define which database alias to use for the `DATABASES` setting.

Parameters **alias** – which database alias to use

The other parameters have the following default values:

Parameters

- **environ** – `True`
- **environ_name** – `DATABASE_URL`
- **environ_prefix** – `None`

```
DATABASES = values.DatabaseURLValue('postgres://myuser@localhost/mydb')
```

class **CacheURLValue**(*default*[, *alias='default'*, *environ=True*, *environ_name='CACHE_URL'*, *environ_prefix=None*])

A *Value* subclass that uses the django-cache-url app to convert a cache configuration value stored in the `CACHE_URL` environment variable into an appropriate setting value. It's inspired by the Twelve-Factor methodology.

By default this *Value* subclass looks for the `CACHE_URL` environment variable.

Takes an optional `alias` parameter to define which database alias to use for the `CACHES` setting.

Parameters **alias** – which cache alias to use

The other parameters have the following default values:

Parameters

- **environ** – `True`
- **environ_name** – `CACHE_URL`
- **environ_prefix** – `None`

```
CACHES = values.CacheURLValue('memcached://127.0.0.1:11211/')
```

class **EmailURLValue**(*default*[, *environ=True*, *environ_name='EMAIL_URL'*, *environ_prefix=None*])

A *Value* subclass that uses the dj-email-url app to convert an email configuration value stored in the EMAIL_URL environment variable into the appropriate settings. It's inspired by the Twelve-Factor methodology.

By default this *Value* subclass looks for the EMAIL_URL environment variable.

Note: This is a special value since email settings are divided into many different settings variables. dj-email-url supports all options though and simply returns a nested dictionary of settings instead of just one setting.

The parameters have the following default values:

Parameters

- **environ** – True
- **environ_name** – EMAIL_URL
- **environ_prefix** – None

```
EMAIL = values.EmailURLValue('console://')
```

class **SearchURLValue**(*default*[, *environ=True*, *environ_name='SEARCH_URL'*, *environ_prefix=None*])

New in version 0.8.

A *Value* subclass that uses the dj-search-url app to convert a search configuration value stored in the SEARCH_URL environment variable into the appropriate settings for use with Haystack. It's inspired by the Twelve-Factor methodology.

By default this *Value* subclass looks for the SEARCH_URL environment variable.

Takes an optional alias parameter to define which search backend alias to use for the HAYSTACK_CONNECTIONS setting.

Parameters **alias** – which cache alias to use

The other parameters have the following default values:

Parameters

- **environ** – True
- **environ_name** – SEARCH_URL
- **environ_prefix** – None

```
HAYSTACK_CONNECTIONS = values.SearchURLValue('elasticsearch://127.0.0.1:9200/my-
↪index')
```

Other values

class **BackendsValue**

A *ListValue* subclass that validates the given list of dotted import paths by trying to import them. In other words, this checks if the backends exist.

```
MIDDLEWARE_CLASSES = values.BackendsValue([
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
```

```
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
])
```

class SecretValue

A *Value* subclass that doesn't allow setting a default value during instantiation and force-enables the use of an environment variable to reduce the risk of accidentally storing secret values in the settings file.

Raises `ValueError` when given a default value

Changed in version 1.0: This value class has the `environ_required` parameter turned to `True`.

```
SECRET_KEY = values.SecretValue()
```

Value mixins

class CastingMixin

A mixin to be used with one of the *Value* subclasses that requires a `caster` class attribute of one of the following types:

- dotted import path, e.g. `'mysite.utils.custom_caster'`
- a callable, e.g. `int`

Example:

```
class TemparatureValue(CastingMixin, Value):
    caster = 'mysite.temperature.fahrenheit_to_celcius'
```

Optionally it can take a `message` class attribute as the error message to be shown if the casting fails. Additionally an `exception` parameter can be set to a single or a tuple of exception classes that are required to be handled during the casting.

class ValidationMixin

A mixin to be used with one of the *Value* subclasses that requires a `validator` class attribute of one of the following types: The validator should raise Django's `ValidationError` to indicate a failed validation attempt.

- dotted import path, e.g. `'mysite.validators.custom_validator'`
- a callable, e.g. `bool`

Example:

```
class TemparatureValue(ValidationMixin, Value):
    validator = 'mysite.temperature.is_valid_temparature'
```

Optionally it can take a `message` class attribute as the error message to be shown if the validation fails.

class MultipleMixin

A mixin to be used with one of the *Value* subclasses that enables the return value of the *to_python* to be interpreted as a dictionary of settings values to be set at once, instead of using the return value to just set one setting.

A good example for this mixin is the *EmailURLValue* value which requires setting many `EMAIL_*` settings.

Cookbook

Calling a Django management command

New in version 0.9.

If you want to call a Django management command programmatically, say from a script outside of your usual Django code, you can use the equivalent of Django's `call_command` function with django-configurations, too.

Simply import it from `configurations.management` instead:

```
from configurations.management import call_command

call_command('dumpdata', exclude=['contenttypes', 'auth'])
```

Envdir

envdir is an effective way to set a large number of environment variables at once during startup of a command. This is great in combination with django-configuration's *`Value`* subclasses when enabling their ability to check environment variables for override values.

Imagine for example you want to set a few environment variables, all you have to do is to create a directory with files that have capitalized names and contain the values you want to set.

Example:

```
$ tree mysite_env/
mysite_env/
- DJANGO_SETTINGS_MODULE
- DJANGO_DEBUG
- DJANGO_DATABASE_URL
- DJANGO_CACHE_URL
- PYTHONSTARTUP

0 directories, 3 files
$ cat mysite_env/DJANGO_CACHE_URL
redis://user@host:port/1
$
```

Then, to enable the `mysite_env` environment variables, simply use the `envdir` command line tool as a prefix for your program, e.g.:

```
$ envdir mysite_env python manage.py runserver
```

See envdir documentation for more information, e.g. using envdir from Python instead of from the command line.

Project templates

You can use a special Django project template that is a copy of the one included in Django 1.5.x and 1.6.x. The following examples assumes you're using pip to install packages.

Django 1.8.x

First install Django 1.8.x and django-configurations:

```
$ pip install -r https://raw.github.com/jazzband/django-configurations/templates/1.8.
↪x/requirements.txt
```

Or Django 1.8:

```
$ django-admin.py startproject mysite -v2 --template https://github.com/jazzband/
↪django-configurations/archive/templates/1.8.x.zip
```

Now you have a default Django 1.8.x project in the `mysite` directory that uses django-configurations.

See the repository of the template for more information:

> https://github.com/jazzband/django-configurations/tree/templates/1.8.x

Celery

< 3.1

Given Celery's way to load Django settings in worker processes you should probably just add the following to the **beginning** of your settings module:

```
import configurations
configurations.setup()
```

That has the same effect as using the `manage.py` or `wsgi.py` utilities. This will also call `django.setup()`.

>= 3.1

In Celery 3.1 and later the integration between Django and Celery has been simplified to use the standard Celery Python API. Django projects using Celery are now advised to add a `celery.py` file that instantiates an explicit `Celery` client app.

Here's how to integrate django-configurations following the example from Celery's documentation:

```
from __future__ import absolute_import

import os

from celery import Celery
from django.conf import settings

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
os.environ.setdefault('DJANGO_CONFIGURATION', 'MySiteConfiguration')

import configurations
configurations.setup()

app = Celery('mysite')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)

@app.task(bind=True)
def debug_task(self):
    print('Request: {0!r}'.format(self.request))
```

iPython notebooks

New in version 0.6.

To use django-configurations with IPython's great notebooks, you have to enable an extension in your IPython configuration. See the IPython documentation for how to create and manage your IPython profile correctly.

Here's a quick how-to in case you don't have a profile yet. Type in your command line shell:

```
$ ipython profile create
```

Then let IPython show you where the configuration file `ipython_config.py` was created:

```
$ ipython locate profile
```

That should print a directory path where you can find the `ipython_config.py` configuration file. Now open that file and extend the `c.InteractiveShellApp.extensions` configuration value. It may be commented out from when IPython created the file or it may not exist in the file at all. In either case make sure it's not a Python comment anymore and reads like this:

```
# A list of dotted module names of IPython extensions to load.
c.InteractiveShellApp.extensions = [
    # .. your other extensions if available
    'configurations',
]
```

That will tell IPython to load django-configurations correctly on startup. It also works with django-extensions's shell_plus management command.

FastCGI

In case you use FastCGI for deploying Django (you really shouldn't) and aren't allowed to use Django's runfcgi management command (that would automatically handle the setup for your if you've followed the quickstart guide above), make sure to use something like the following script:

```
#!/usr/bin/env python

import os

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
os.environ.setdefault('DJANGO_CONFIGURATION', 'MySiteConfiguration')

from configurations.fastcgi import runfastcgi

runfastcgi(method='threaded', daemonize='true')
```

As you can see django-configurations provides a helper module `configurations.fastcgi` that handles the setup of your configurations.

Sphinx

In case you would like to user the amazing *autodoc* feature of the documentation tool Sphinx, you need add django-configurations to your `extensions` config variable and set the environment variable accordingly:

```
# My custom Django environment variables
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
os.environ.setdefault('DJANGO_CONFIGURATION', 'Dev')

# Add any Sphinx extension module names here, as strings. They can be extensions
# coming with Sphinx (named 'sphinx.ext.*') or your custom ones.
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.intersphinx',
    'sphinx.ext.viewcode',
    # ...
    'configurations.sphinx',
]

# ...
```

Changed in version 2.0.

Please note that the sphinx callable has been moved from `configurations` to `configurations.sphinx`.

Changelog

v2.0 (2016-07-29)

- **BACKWARD INCOMPATIBLE** Drop support of Python 2.6 and 3.2
- **BACKWARD INCOMPATIBLE** Drop support of Django < 1.8
- **BACKWARD INCOMPATIBLE** Moved sphinx callable has been moved from `configurations` to `configurations.sphinx`.
- **BACKWARD INCOMPATIBLE** Removed the previously deprecated `configurations.Settings` class in favor of the `configurations.Configuration` added in 0.4. This removal was planned for the 1.0 release and is now finally enacted.
- Add multiprocessing support for sphinx integration
- Fix a RemovedInDjango19Warning warning

v1.0 (2016-01-04)

- Project has moved to Jazzband. See guidelines for contributing.
- Support for Django 1.8 and above.
- Allow `Value` classes to be used outside of `Configuration` classes. (#62)
- Fixed "Value with ValidationMixin will raise ValueError if no default assigned". (#69)
- Fixed wrong behaviour when assigning BooleanValue. (#83)
- Add ability to programmatically call Django commands from configurations using `call_command`.
- Added SingleNestedTupleValue and SingleNestedListValue classes. (#85)
- Several other miscellaneous bugfixes.

v0.8 (2014-01-16)

- Added `SearchURLValue` to configure Haystack `HAYSTACK_CONNECTIONS` settings.

v0.7 (2013-11-26)

- Removed the broken stdout wrapper that displayed the currently enabled configuration when using the runserver management command. Added a logging based solution instead.
- Fixed default value of `CacheURLValue` class that was shadowed by an unneeded name parameter. Thanks to Stefan Wehrmeyer.
- Fixed command line options checking in the importer to happen before the validation. Thanks to Stefan Wehrmeyer.
- Added Tox test configuration.
- Fixed an erroneous use of `PathValue` in the 1.6.x project template.

v0.6 (2013-09-19)

- Added a IPython extension to support IPython notebooks correctly. See the *Cookbook* for more information.

v0.5.1 (2013-09-12)

- Prevented accidentally parsing the command line options to look for the `--configuration` option outside of Django's management commands. This should fix a problem with gunicorn's own `--config` option. Thanks to Brian Rosner for the report.

v0.5 (2013-09-09)

- Switched from raising Django's `ImproperlyConfigured` exception on errors to standard `ValueError` to prevent hiding those errors when Django specially handles the first.
- Switched away from d2to1 as a way to define package metadata since distutils2 is dead.
- Extended `Value` class documentation and fixed other issues.
- Moved tests out of the `configurations` package for easier maintenance.

v0.4 (2013-09-03)

- Added `Value` classes and subclasses for easier handling of settings values, including populating them from environment variables.
- Renamed `configurations.Settings` class to `configurations.Configuration` to better describe what the class is all about. The old class still exists and is marked as pending deprecation. It'll be removed in version 1.0.
- Added a `setup` method to handle the new `Value` classes and allow an in-between modification of the configuration values.
- Added Django project templates for 1.5.x and 1.6.x.
- Reorganized and extended documentation.

v0.3.2 (2014-01-16)

- Fixed an installation issue.

v0.3.1 (2013-09-20)

- Backported a fix from master that makes 0.3.x compatible with newer versions of six.

v0.3 (2013-05-15)

- Added `pristinemethod` decorator to be able to have callables as settings.
- Added `pre_setup` and `post_setup` method hooks to be able to run code before or after the settings loading is finished.
- Minor docs and tests cleanup.

v0.2.1 (2013-04-11)

- Fixed a regression in parsing the new `-C`/`--configuration` management command option.
- Minor fix in showing the configuration in the `runserver` management command output.

v0.2 (2013-03-27)

- **backward incompatible change** Dropped support for Python 2.5! Please use the 0.1 version if you really want.
- Added Python>3.2 and Django 1.5 support!
- Catch error when getting or evaluating callable setting class attributes.
- Simplified and extended tests.
- Added optional `-C`/`--configuration` management command option similar to Django's `--settings` option
- Fixed the runserver message about which setting is used to show the correct class.
- Stopped hiding AttributeErrors happening during initialization of settings classes.
- Added FastCGI helper.
- Minor documentation fixes

v0.1 (2012-07-21)

- Initial public release

CHAPTER 6

Alternatives

Many thanks to those project that have previously solved these problems:

- The Pinax project for spearheading the efforts to extend the Django project metaphor with reusable project templates and a flexible configuration environment.
- django-classbasedsettings by Matthew Tretter for being the immediate inspiration for django-configurations.

CHAPTER 7

Bugs and feature requests

As always your mileage may vary, so please don't hesitate to send feature requests and bug reports:

https://github.com/jazzband/django-configurations/issues

Thanks!

Python Module Index

C

configurations.values, 14

Index

B

BackendsValue (class in configurations.values), 21
BooleanValue (class in configurations.values), 17

C

CacheURLValue (class in configurations.values), 20
CastingMixin (class in configurations.values), 22
configurations.values (module), 14

D

DatabaseURLValue (class in configurations.values), 20
DecimalValue (class in configurations.values), 17
DictValue (class in configurations.values), 19

E

EmailURLValue (class in configurations.values), 20
EmailValue (class in configurations.values), 19

F

FloatValue (class in configurations.values), 17

I

IntegerValue (class in configurations.values), 17
IPValue (class in configurations.values), 19

L

ListValue (class in configurations.values), 17

M

MultipleMixin (class in configurations.values), 22

P

PathValue (class in configurations.values), 19

R

RegexValue (class in configurations.values), 19

S

SearchURLValue (class in configurations.values), 21
SecretValue (class in configurations.values), 22
setup() (Value method), 16
SetValue (class in configurations.values), 19
SingleNestedListValue (class in configurations.values), 18
SingleNestedTupleValue (class in configurations.values), 18

T

to_python() (Value method), 16
TupleValue (class in configurations.values), 18

U

URLValue (class in configurations.values), 19

V

ValidationMixin (class in configurations.values), 22
Value (class in configurations.values), 16