django-configurations Documentation

Release dev

Jannis Leidel

September 03, 2013

CONTENTS

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.

CHAPTER

ONE

QUICKSTART

Install django-configurations:

```
pip install django-configurations
```

Then subclass the included `configurations.Settings` 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 Settings

class Dev(Settings):
    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 9 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.

CHAPTER

TWO

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

THREE

OKAY, HOW DOES IT WORK?

Any subclass of the `configurations.Settings` 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

FOUR

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

FIVE

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.

5.1 Server specific settings

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

```
from configurations import Settings

class Base(Settings):
    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
```

5.2 Global settings defaults

Every `configurations.Settings` 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 Settings

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

    @property
    def LANGUAGES(self):
        return Settings.LANGUAGES + (('tlh', 'Klingon'),)
```

5.3 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 Settings class:

```
from configurations import Settings

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

5.4 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 Settings, pristinemethod

class Prod(Settings):

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

Lambdas work, too:

```
from configurations import Settings, pristinemethod

class Prod(Settings):
    ACCESS_FUNCTION = pristinemethod(lamda user: user.is_staff)
```

5.5 Setup methods

New in version 0.3. If there is something required to be set up before or after the settings loading happens, please override the `pre_setup` or `post_setup` class methods like so (don't forget to apply the Python `@classmethod` decorator:

```
from configurations import Settings

class Prod(Settings):
    # ...

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

    @classmethod
    def post_setup(cls):
        print("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 Settings

class Prod(Settings):
    # ...

    @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.

CHAPTER

SIX

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

SEVEN

COOKBOOK

7.1 Celery

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

```
from configurations import importer
importer.install()
```

That has the same effect as using the `manage.py` or `wsgi.py` utilities mentioned above.

7.2 FastCGI

In case you use FastCGI for deploying Django (you really shouldn't) and aren't allowed to us 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
import sys

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

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.

CHAPTER

EIGHT

BUGS AND FEATURE REQUESTS

As always you mileage may vary, so please don't hesitate to send in feature requests and bug reports at the usual place:

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

Thanks!

CHAPTER

NINE

CHANGELOG

9.1 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.

9.2 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.

9.3 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

9.4 v0.1 (2012-07-21)

- Initial public release