django-appconf Documentation

Release dev

Jannis Leidel and individual contributors

January 28, 2013

CONTENTS

A helper class for handling configuration defaults of packaged Django apps gracefully.

CHAPTER

ONE

OVERVIEW

Say you have an app called `myapp` with a few defaults, which you want to refer to in the app's code without repeating yourself all the time. `appconf` provides a simple class to implement those defaults. Simply add something like the following code somewhere in your app files:

```
from appconf import AppConf

class MyAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )
```

Note: `AppConf` classes depend on being imported during startup of the Django process. Even though there are multiple modules loaded automatically, only the `models` modules (usually the `models.py` file of your app) are guaranteed to be loaded at startup. Therefore it's recommended to put your `AppConf` subclass(es) there, too.

The settings are initialized with the capitalized app label of where the setting is located at. E.g. if your `models.py` with the `AppConf` class is in the `myapp` package, the prefix of the settings will be `MYAPP`.

You can override the default prefix by specifying a `prefix` attribute of an inner `Meta` class:

```
from appconf import AppConf

class AcmeAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )

    class Meta:
        prefix = 'acme'
```

The `MyAppConf` class will automatically look at Django's global settings to determine if you've overridden it. For example, adding this to your site's `settings.py` would override `SETTING_1` of the above `MyAppConf`:

```
ACME_SETTING_1 = "uno"
```

In case you want to use a different settings object instead of the default `'django.conf.settings'`, set the `holder` attribute of the inner `Meta` class to a dotted import path:

```
from appconf import AppConf

class MyAppConf(AppConf):
```

```
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )

    class Meta:
        prefix = 'acme'
        holder = 'acme.conf.settings'
```

If you ship an `AppConf` class with your reusable Django app, it's recommended to put it in a `conf.py` file of your app package and import `django.conf.settings` in it, too:

```
from django.conf import settings
from appconf import AppConf

class MyAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )
```

In the other files of your app you can easily make sure the settings are correctly loaded if you import Django's settings object from that module, e.g. in your app's `views.py`:

```
from django.http import HttpResponse
from myapp.conf import settings

def index(request):
    text = 'Setting 1 is: %s' % settings.MYAPP_SETTING_1
    return HttpResponse(text)
```

CHAPTER

TWO

INSTALLATION

Install django-appconf with your favorite Python package manager, e.g.:

```
pip install django-appconf
```

CHAPTER

THREE

CONTENTS

3.1 Usage

It's strongly recommended to use the usual `from django.conf import settings` in your own code to access the configured settings.

But you can also **OPTIONALLY** use your app's own settings object directly, by instantiating it in place:

```
from myapp.models import MyAppConf

myapp_settings = MyAppConf()

print myapp_settings.SETTING_1
```

Note that accessing the settings that way means they don't have a prefix.

`AppConf` instances don't automatically work as proxies for the global settings. But you can enable this if you want by setting the `proxy` attribute of the inner `Meta` class to `True`:

```
from appconf import AppConf

class MyAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )

    class Meta:
        proxy = True

myapp_settings = MyAppConf()

if "myapp" in myapp_settings.INSTALLED_APPS:
    print "yay, myapp is installed!"
```

In case you want to override some settings programmatically, you can simply pass the value when instantiating the `AppConf` class:

```
from myapp.models import MyAppConf

myapp_settings = MyAppConf(SETTING_1='something completely different')

if 'different' in myapp_settings.SETTINGS_1:
    print "yay, I'm different!"
```

3.1.1 Custom configuration

Each of the settings can be individually configured with callbacks. For example, in case a value of a setting depends on other settings or other dependencies. The following example sets one setting to a different value depending on a global setting:

```
from django.conf import settings
from appconf import AppConf

class MyCustomAppConf(AppConf):
    ENABLED = True

    def configure_enabled(self, value):
        return value and not settings.DEBUG
```

The value of `MYAPP_ENABLED` will vary depending on the value of the global `DEBUG` setting.

Each of the app settings can be customized by providing a method `configure_<lower_setting_name>` that takes the default value as defined in the class attributes of the `AppConf` subclass or the override value from the global settings as the only parameter. The method **must return** the value to be use for the setting in question.

After each of the `*_configure` methods have been called, the `AppConf` class will additionally call a main `configure` method, which can be used to do any further custom configuration handling, e.g. if multiple settings depend on each other. For that a `configured_data` dictionary is provided in the setting instance:

```
from django.conf import settings
from appconf import AppConf

class MyCustomAppConf(AppConf):
    ENABLED = True
    MODE = 'development'

    def configure_enabled(self, value):
        return value and not settings.DEBUG

    def configure(self):
        mode = self.configured_data['MODE']
        enabled = self.configured_data['ENABLED']
        if not enabled and mode != 'development':
            print "WARNING: app not enabled in %s mode!" % mode
        return self.configured_data
```

Note: Don't forget to return the configured data in your custom `configure` method if you edit it.

3.2 Reference

class `appconf.`**AppConf**

A representation of a template tag. For example:

```
class MyAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )
```

configure_*(value)
Method for each of the app settings for custom configuration which gets the value passed of the class attribute or the appropriate override value of the `holder` settings, e.g.:

```
class MyAppConf(AppConf):
    DEPLOYMENT_MODE = "dev"

    def configure_deployment_mode(self, value):
        if on_production():
            value = "prod"
        return value
```

The method **must return** the value to be use for the setting in question.

class `AppConf`**.Meta**
An `AppConf` takes options via a `Meta` inner class:

```
class MyAppConf(AppConf):
    SETTING_1 = "one"
    SETTING_2 = (
        "two",
    )

    class Meta:
        proxy = False
        prefix = 'myapp'
        required = ['SETTING_3', 'SETTING_4']
        holder = 'django.conf.settings'
```

prefix
Explicitly choose a prefix for all settings handled by the `AppConf` class. If not given, the prefix will be the capitalized class module name.

For example, `acme` would turn into settings like `ACME_SETTING_1`.

required
A list of settings that must be defined. If any of the specified settings are not defined, `ImproperlyConfigured` will be raised. New in version 0.6.

holder
The global settings holder to use when looking for overrides and when setting the configured values.

Defaults to `'django.conf.settings'`.

proxy
A boolean, if set to `True` will enable proxying attribute access to the `holder`.

3.3 Changelog

3.3.1 0.6 (2013-01-28)

- Added `required` attribute to `Meta` to be able to specify which settings are required to be set.
- Moved to Travis for the tests: http://travis-ci.org/jezdez/django-appconf
- Stopped support for Django 1.2.X.
- Introduced support for Python >= 3.2.

3.3.2 0.5 (2012-02-20)

- Install as a package instead of a module.
- Refactored tests to use django-jenkins for enn.io's CI server.

3.3.3 0.4.1 (2011-09-09)

- Fixed minor issue in installation documentation.

3.3.4 0.4 (2011-08-24)

- Renamed `app_label` attribute of the inner `Meta` class to `prefix`. The old form `app_label` will work in the meantime.
- Added `holder` attribute to the inner `Meta` class to be able to specify a custom "global" setting holder. Default: "'django.conf.settings"'
- Added `proxy` attribute to the inner `Meta` class to enable proxying of `AppConf` instances to the settings holder, e.g. the global Django settings.
- Fixed issues with `configured_data` dictionary available in the `configure` method of `AppConf` classes with regard to subclassing.

3.3.5 0.3 (2011-08-23)

- Added tests with 100% coverage.
- Added ability to subclass `Meta` classes.
- Fixed various bugs with subclassing and configuration in subclasses.

3.3.6 0.2.2 (2011-08-22)

- Fixed another issue in the `configure()` API.

3.3.7 0.2.1 (2011-08-22)

- Fixed minor issue in `configure()` API.

3.3.8 0.2 (2011-08-22)

- Added `configure()` API to `AppConf` class which is called after configuring each setting.

3.3.9 0.1 (2011-08-22)

- First public release.