django-configglue Documentation

Release

Ricardo Kirkner

August 05, 2011

CONTENTS

Contents:

CHAPTER

ONE

DJANGO-CONFIGGLUE 101

This is a minimalistic step-by-step guide on how to start using django-configglue to manage the settings for your Django project.

1.1 Create the schema

First we need to create the schema that will define the settings we want to support in our configuration files.

Start by creating a module called *schema.py*, such as

```
import django
from configglue import schema
from django_configglue.schema import schemas


DjangoSchema = schemas.get(django.get_version())

class MySchema(DjangoSchema):
    foo = schema.IntOption()
    bar = schema.BoolOption()
```

The *MySchema* schema will support all Django supported settings (as defined in the *DjangoSchema* schema), and it introduces two custom options (*foo* and *bar* in the default section – *__main__*).

1.2 Create the config files

Now we need to create the configuration file where we specify the values we want to have for our options. Create a file called *main.cfg*

```
[__main__]
foo = 1
bar = true

[django]
database_engine = sqlite3
database_name = :memory:
installed_apps =
    django.contrib.auth
    django.contrib.contenttypes
    django.contrib.sessions
```

```
    django.contrib.sites
    django_configglue
```

1.3 Glue into django

Finally, we need to implement the glue between configglue and Django, so that it can read out the settings defined in our configuration files.

Replace the standard *settings.py* module in your project with

```
from django_configglue.utils import configglue
from .schema import MySchema

# make django aware of configglue-based configuration
configglue(MySchema, ['main.cfg'], __name__)
```

1.4 Test

And let's make sure everything works as expected

```
$ python manage.py settings --validate
Settings appear to be fine.
```

1.5 Profit!

That's it! Your project now uses django-configglue to manage it's configuration. Congratulations!

If you want to know more about django-configglue, read *Using django-configglue: a more in-depth walkthrough*.

CHAPTER

TWO

USING DJANGO-CONFIGGLUE: A MORE IN-DEPTH WALKTHROUGH

Warning: This walkthrough assumes previous knowledge of configglue, and how to read and write configglue based configuration files.

2.1 The basics

Let's start by creating a Django project.

```
django-admin startproject quickstart
```

In order for Django to be aware of the configuration glue provided by django-configglue, we need to replace the standard *settings.py* module with a version that knows how to read the settings from the ini-style configuration files.

```
cd quickstart
mv settings.py settings.py.orig
```

Replace *settings.py* with

```
import django
from django_configglue.schema import schemas
from django_configglue.utils import configglue

DjangoSchema = schemas.get(django.get_version())

# make django aware of configglue-based configuration
configglue(DjangoSchema, ['main.cfg', 'local.cfg'], __name__)
```

This code snippet defines a schema (a static description of which configuration options and sections are available, including a type definition for each option, in order to allow for configuration validation) for Django's settings, which it will use to create a parser that knows how to parse these kind of configuration files. It will then read two files, called *main.cfg* and *local.cfg*, in that order, and will populate the module's local dictionary with the parsed values.

As we decided to read *main.cfg* and *local.cfg* files if available, let's create the *main.cfg* file first. This file is where you define the default settings and their values. You can later override those using *local.cfg* or any other file, as we'll see later.

Put the following content into a file called *main.cfg*

```
[django]
installed_apps = django_configglue
```

This is the smallest Django configuration file required for django-configglue to work, equivalent to a standard Django *settings.py* module with the following content

```
INSTALLED_APPS = ('django_configglue',)
```

Let's make sure this configuration is valid, meaning it's well formed, and should work alright.

```
$ python manage.py settings --validate
Settings appear to be fine.
```

Great, everything seems to be in place.

Now, we can query configglue to see what configuration options have been defined.

```
$ python manage.py settings --show
ROOT_URLCONF = 'urls'
SETTINGS_MODULE = 'quickstart.settings'
SITE_ID = 1
```

While this output might look a bit odd (where are the configuration settings we defined? why are these other settings listed, which we never specified?), everything can be explained.

The *–show* parameter lists any setting not defined in the *global_settings.py* module (that is part of Django). To include those settings, you have to pass in the *–global* option as well

```
$ python manage.py settings --show --global
...
INSTALLED_APPS = ['django_configglue']
...
```

This lists all settings, including the ones that we defined previously in our configuration file.

Since the list can be quite extensive, there are better ways for getting the values we care for. It's possible to request just the settings we're interested in.

```
$ python manage.py settings email_port
EMAIL_PORT = 25
$ python manage.py settings EMAIL_PORT
EMAIL_PORT = 25
```

Tip: The name of the setting is pseudo-case-insensitive, as django-configglue will search for the name as provided, and fall back to using it's upper-case version before failing.

As can be seen, the output of the settings command lists the settings in Django's standard format.

If we pass in a non-existing setting, we'll get an error message

```
$ python manage.py settings email
setting EMAIL not found
```

An interesting option is *–locate*, which will make more sense once we cover layered configuration. This option displays the location where the requested setting was last defined.

```
$ python manage.py settings --locate database_name
setting DATABASE_NAME last defined in '/path/to/main.cfg'
```

2.2 Layered configuration

Configglue supports reading multiple configuration files so that specific settings can be grouped into different files, and overridden as needed.

In the *settings.py* module previously defined, we specified that django-configglue should read the configuration values from the following files (and in order):

1. main.cfg
2. local.cfg

The idea is to have the default values defined in the *main.cfg* file, and override as appropriate using the *local.cfg* file. The goal of splitting the configuration this way is to be able to hold part of the configuration in version control (*main.cfg*), and other aspects not (*local.cfg*). The contents of the *local.cfg* file would follow the same syntax as for the *main.cfg* file, but as they are read in later, would override any value already defined in *main.cfg*.

There is another way of telling configglue to read in other files, without having to specify them in the *settings.py* module. This is achieved by means of the **includes** keyword.

For example, if you add this to *main.cfg*

```
[__main__]
includes = custom.cfg
```

Note: The __main__ section is a special section that is always present, independently of any section defined by the schema.

---

and create a *custom.cfg* file with the following content:

```
[django]
installed_apps =
    django.contrib.auth
    django.contrib.contenttypes
    django.contrib.sessions
    django.contrib.sites
    django_configglue
```

The *INSTALLED_APPS* setting will be read from the *custom.cfg* configuration file, as can be verified by running

```
$ python manage settings.py installed_apps
INSTALLED_APPS = ['django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'd
```

and

```
$ python manage settings.py --locate installed_apps
setting INSTALLED_APPS last defined in '/path/to/custom.cfg'
```

This last command shows that the *INSTALLED_APPS* setting was effectively read from the *custom.cfg* file.

2.3 Interpolation

Another interesting feature of configglue, which django-configglue based configuration files can therefore also use, is variable interpolation. This means that a variable can be defined in terms of another variable.

If we add the following snippet to the 'django' section of the *custom.cfg* configuration file

```
template_debug = %(debug)s
```

We can then verify that the TEMPLATE_DEBUG setting value depends on the value of the *DEBUG* setting.

```
$ python manage.py settings debug
DEBUG = True
$ python manage.py settings template_debug
TEMPLATE_DEBUG = True
```

Even if we change the value of the *DEBUG* setting (go ahead and add the following to the *custom.cfg* file, under the 'django' section)

```
debug = false
```

and then issue the command

```
$ python manage.py settings template_debug debug
TEMPLATE_DEBUG = False
DEBUG = False
```

2.4 Command line integration

So far we've looked at statically-defined configuration values. One of the real benefits of configglue is being able to override variables by means of command-line provided parameters.

Let's look at one parameter called INTERNAL_IPS

```
$ python manage.py settings internal_ips
INTERNAL_IPS = []
```

and let's override that setting from the command-line

```
$ python manage.py settings --django_internal_ips=127.0.0.1,192.168.0.1 internal_ips
INTERNAL_IPS = ['127.0.0.1', '192.168.0.1']
```

As can be seen, the way to specify an option from the command line is to specify

```
--<section>_<option>=<value>
```

Take care that the specified value has to be valid according the the option's type, as defined by it's schema, as it will be casted to match it.

In this example, the type for *INTERNAL_IPS* is a *TupleOption*, so the value will be interpreted as a tuple of strings, separated by commas.

2.5 Validation

Finally, one of the key benefits of using django-configglue for managing your Django settings is the ability to validate the configuration before restarting your server.

Validation will ensure all required parameters have been assigned values, and that there are no unknown sections mentioned in the configuration files (useful to catch typos). Also, type validation will take place, ensuring the values used for each option are valid according to that option's type as specified by the schema.

So, generally, a valid configuration will produce the following result:

```
$ python manage.py settings --validate
Settings appear to be fine.
```

However, if you try to specify an invalid value for some option, the corresponding error will be raised

```
$ python manage.py settings --django_site_id=foo
Traceback (most recent call last):
...
ValueError: Invalid value 'foo' for IntOption 'site_id' in section 'django'. Original exception was:
```

If, on the other hand, an invalid section name is used that will be reported too. Edit the *custom.cfg* file so that the section name reads

```
[dajngo]
```

instead of

```
[django]
```

When validating the configuration

```
$ python manage.py settings --validate
Error: Settings did not validate againt schema.

Sections in configuration are missing from schema: dajngo
```

it will be noted that the *dajngo* section is not valid according to the schema used.