django-dynamic-preferences Documentation

Release 1.0

Eliot Berriot

February 21, 2017

Contents

Dynamic-preferences is a Django app, BSD-licensed, designed to help you manage your project settings. While most of the time, a *settings.py* file is sufficient, there are some situations where you need something more flexible such as:

- per-user settings (or, generally speaking, per instance settings)
- settings change without server restart

For per-instance settings, you could actually store them in some kind of profile model. However, it means that every time you want to add a new setting, you need to add a new column to the profile DB table. Not very efficient.

Dynamic-preferences allow you to register settings (a.k.a. preferences) in a declarative way. Preferences values are serialized before storage in database, and automatically deserialized when you need them.

With dynamic-preferences, you can update settings on the fly, through django's admin or custom forms, without restarting your application.

The project is tested and work under Python 2.7 and 3.4, with django >=1.7.

CHAPTER 1

Features

- Simple to setup
- Admin integration
- Forms integration
- Bundled with global and per-user preferences
- Can be extended to other models if need (e.g. per-site preferences)
- Integrates with django caching mechanisms to improve performance

If you're still interested, head over Installation.

Contents:

Installation

Dynamic-preferences is available on PyPI and can be installed with:

```
pip install django-dynamic-preferences
```

Setup

Add this to your `settings.INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # ...
    'django.contrib.auth',
    'dynamic_preferences',
    # comment the following line if you don't want to use user preferences
    'dynamic_preferences.users.apps.UserPreferencesConfig',
)
```

Then, create missing tables in your database:

```
python manage.py migrate dynamic_preferences
```

Add this to `settings.TEMPLATE_CONTEXT_PROCESSORS` if you want to access preferences from templates:

```
TEMPLATE_CONTEXT_PROCESSORS =  (
    'django.core.context_processors.request',
    'dynamic_preferences.processors.global_preferences',
)
```

Settings

Also, take some time to look at provided settings if you want to customize the package behaviour:

```
# available settings with their default values
DYNAMIC_PREFERENCES = {

    # a python attribute that will be added to model instances with preferences
    # override this if the default collide with one of your models attributes/fields
    'MANAGER_ATTRIBUTE': 'preferences',

    # The python module in which registered preferences will be searched within each app
    'REGISTRY_MODULE': 'dynamic_preferences_registry',

    # Allow quick editing of preferences directly in admin list view
    # WARNING: enabling this feature can cause data corruption if multiple users
    # use the same list view at the same time, see https://code.djangoproject.com/ticket/11313
    'ADMIN_ENABLE_CHANGELIST_FORM': False,

    # Customize how you can access preferences from managers. The default is to
    # separate sections and keys with two underscores. This is probably not a settings you'll
    # want to change, but it's here just in case
    'SECTION_KEY_SEPARATOR': '__',

    # Use this to disable caching of preference. This can be useful to debug things
    'ENABLE_CACHE': True,

    # Use this to disable checking preferences names. This can be useful to debug things
    'VALIDATE_NAMES': True,
}
```

Quickstart

Glossary

Preference An object that deals with preference logic, such as serialization, deserialization, form display, default values, etc. After being defined, preferences can be tied via registries to one ore many preference models, which will deal with database persistence.

PreferenceModel A model that store preferences values in database. A preference model may be tied to a particular model instance, which is the case for UserPreferenceModel, or concern the whole project, as GlobalPreferenceModel.

Create and register your own preferences

In this example, we assume you are building a blog. Some preferences will apply to your whole project, while others will belong to specific users.

First, create a *dynamic_preferences_registry.py* file within one of your project app. The app must be listed in `settings.INSTALLED_APPS`.

Let's declare a few preferences in this file:

```
# blog/dynamic_preferences_registry.py

from dynamic_preferences.types import BooleanPreference, StringPreference, Section
from dynamic_preferences.registries import global_preferences_registry
from dynamic_preferences.users.registries import user_preferences_registry

# we create some section objects to link related preferences together

general = Section('general')
discussion = Section('discussion')

# We start with a global preference
@global_preferences_registry.register
class SiteTitle(StringPreference):
    section = general
    name = 'title'
    default = 'My site'

@global_preferences_registry.register
class MaintenanceMode(BooleanPreference):
    name = 'maintenance_mode'
    default = False

# now we declare a per-user preference
@user_preferences_registry.register
class CommentNotificationsEnabled(BooleanPreference):
    """Do you want to be notified on comment publication ?"""
    section = discussion
    name = 'comment_notifications_enabled'
    default = True
```

The `section` attribute is a convenient way to keep your preferences in different... well... sections. While you can totally forget this attribute, it is used in various places like admin or forms to filter and separate preferences. You'll probably find it useful if you have many different preferences. The `name` attribute is a unique identifier for your preference. However, You can share the same name for various preferences if you use different sections.

Important: preferences names and sections names (if you use them) are persisted in database and should be considered as primary keys. If, for some reason, you want to update a preference or section name and keep already persisted preferences sync, you'll have to write a data migration.

Retrieve and update preferences

You can get and update preferences via a `Manager`, a dictionary-like object. The logic is almost exactly the same for global preferences and per-instance preferences.

```
from dynamic_preferences.registries import global_preferences_registry

# We instantiate a manager for our global preferences
global_preferences = global_preferences_registry.manager()

# now, we can use it to retrieve our preferences
```

```
# the lookup for a preference has the following form: <section>__<name>
assert global_preferences['general__title'] == 'My site'

# You can also access section-less preferences
assert global_preferences['maintenance_mode'] == False

# We can update our preferences values the same way
global_preferences['maintenance_mode'] = True
```

For per-instance preferences it's even easier. You can access each instance preferences via the `preferences` attribute.

```
from django.contrib.auth import get_user_model

user = get_user_model().objects.get(username='eliot')

assert user.preferences['discussion__comment_notifications_enabled'] == True

# Disable the notification system
user.preferences['discussion__comment_notifications_enabled'] = False
```

Under the hood

When you access a preference value (e.g. via `global_preferences['maintenance_mode']`), dynamic-preferences follows these steps:

1. It checks for the cached value (using classic django cache mechanisms)
2. If no cache key is found, it queries the database for the value
3. If the value does not exists in database, a new row is added with the default preference value, and the value is returned. The cache is updated to avoid another database query the next time you want to retrieve the value.

Therefore, in the worst-case scenario, accessing a single preference value can trigger up to two database queries. Most of the time, however, dynamic-preferences will only hit the cache.

When you set a preference value (e.g. via‘‘global_preferences['maintenance_mode'] = True‘‘), dynamic-preferences follows these steps:

1. The corresponding row is queried from the database (1 query)
2. The new value is set and persisted in db (1 query)
3. The cache is updated.

Updating a preference value will always trigger two database queries.

Misc methods for retrieving preferences

A few other methods are available on managers to retrieve preferences:

- *manager.all()*: returns a *dict* containing all preferences identifiers and values
- ***manager.by_name()*: returns a *dict* containing all preferences identifiers and values.** The preference section name (if any) is removed from the identifier
- *manager.get_by_name(name)*: returns a single preference value using only the preference name

About serialization

When you get or set preferences values, you interact with Python values. On the database/cache side, values are serialized before storage.

Dynamic preferences handle this for you, using each preference type (BooleanPreference, StringPreference, IntPreference, etc.). It's totally possible to create your own preferences types and serializers, have a look at `types.py` and `serializers.py` to get started.

Admin integration

Dynamic-preferences integrates with *django.contrib.admin* out of the box. You can therefore use the admin interface to edit preferences values, which is particularly convenient for global preferences.

Forms

A form builder is provided if you want to create and update preferences in custom views.

```
from dynamic_preferences.forms import global_preference_form_builder

# get a form for all global preferences
form_class = global_preference_form_builder()

# get a form for global preferences of the 'general' section
form_class = global_preference_form_builder(section='general')

# get a form for a specific set of preferences
# You can use the lookup notation (section__name) as follow
form_class = global_preference_form_builder(preferences=['general__title'])

# or pass explicitly the section and names as an iterable of tuples
form_class = global_preference_form_builder(preferences=[('general', 'title'), ('another_section', 'a
```

Getting a form for a specific instance preferences works similarly, except that you need to provide the user instance:

```
from dynamic_preferences.forms import user_preference_form_builder

form_class = user_preference_form_builder(instance=request.user)
form_class = user_preference_form_builder(instance=request.user, section='discussion')
```

Preferences attributes

You can customize a lof of preferences behaviour some class attributes / methods.

For example, if you want to customize the `verbose_name` of a preference you can simply do:

```
class MyPreference(StringPreference):
    verbose_name = "This is my preference"
```

But if you need more customization, you can do:

```
import datetime

class MyPreference(StringPreference):

```

```
    def get_verbose_name(self):
        return "Verbose name instantiated on {0}".format(datetime.datetime.now())
```

Both methods are perfectly valid. You can override the following attributes:

- `field_class`: the field class used to edit the preference value
- `field_kwargs`: kwargs that are passed to the field class upon instantiation. Ensure to call `super()` since some default are provided.
- `verbose_name`: used in admin and as a label for the field
- `help_text`: used in admin and in the field
- `default`: the default value for the preference, taht will also be used as initial data for the form field
- `widget`: the widget used for the form field

Accessing global preferences within a template

Dynamic-preferences provide a context processors (remember to add them to your settings, as described in "Installation") that will pass global preferences values to your templates:

```
# myapp/templates/mytemplate.html

<title>{{ global_preferences.general__title }}</title>

{% if request.user.preferences.discussion__comment_notifications_enabled %}
    You will receive an email each time a comment is published
{% else %}
    <a href='/subscribe'>Subscribe to comments notifications</a>
{% endif %}
```

Bundled views and urls

Example views and urls are bundled for global and per-user preferences updating. Include this in your URLconf:

```
urlpatterns = [
    # your project urls here
    url(r'^preferences/', include('dynamic_preferences.urls')),
]
```

Then, in your code:

```
from django.core.urlresolvers import reverse

# URL to a page that display a form to edit all global preferences
url = reverse("dynamic_preferences.global")

# URL to a page that display a form to edit global preferences of the general section
url = reverse("dynamic_preferences.global.section", kwargs={'section': 'general'})

# URL to a page that display a form to edit all preferences of the user making the request
url = reverse("dynamic_preferences.user")

# URL to a page that display a form to edit preferences listed under section 'discussion' of the use
url = reverse("dynamic_preferences.user.section", kwargs={'section': 'discussion'})
```

Preferences lifecycle

Update

To do, help welcome :)

Deletion

If you remove preferences from your registry, corresponding data rows won't be deleted automatically.

In order to keep a clean database and delete obsolete rows, you can use the *checkpreferences* management command. This command will check all preferences in database, ensure they match a registered preference class and delete rows that do not match any registerd preference.

> **Warning:** Run this command carefully, since it can lead to data loss.

Bind preferences to arbitrary models

By default, dynamic-preferences come with two kinds of preferences:

- Global preferences, which are not tied to any particular model instance
- User preferences, which apply to a specific user

While this can be enough, your project may require additional preferences. For example, you may want to bind preferences to a specific `Site` instance. Don't panic, dynamic-preferences got you covered.

In order to achieve this, you'll need to follow this process:

1. Create a preference model with a ForeignKey to Site
2. Create a registry to store available preferences for sites

The following guide assumes you want to bind preferences to the `django.contrib.sites.Site` model.

Create a preference model

You'll need to subclass `PerInstancePreferenceModel` model, and add a `ForeignKey` field pointing to the target model:

```
# yourapp/models.py
from django.contrib.sites.models import Site
from dynamic_preferences.models import PerInstancePreferenceModel

class SitePreferenceModel(PerInstancePreferenceModel):

    # note: you *have* to use the `instance` field
    instance = models.ForeignKey(Site)
```

Now, you can create a migration for your newly created model with `python manage.py makemigrations`, apply it with `python manage.py migrate`.

Create a registry to collect your model preferences

Now, you have to create a registry to collect preferences belonging to the `Site` model:

```
# yourapp/registries.py
from dynamic_preferences.registries import PerInstancePreferenceRegistry

class SitePreferenceRegistry(PerInstancePreferenceRegistry):
    pass

site_preferences_registry = SitePreferenceRegistry()
```

Then, you simply have to connect your `SitePreferenceModel` to your registry. You should do that in an `apps.py` file, as follows:

```
# yourapp/apps.py
from django.apps import AppConfig
from django.conf import settings

from dynamic_preferences.registries import preference_models
from .registries import site_preferences_registry

class YourAppConfig(AppConfig):
    name = 'your_app'

    def ready(self):
        SitePreferenceModel = self.get_model('SitePreferenceModel')

        preference_models.register(SitePreferenceModel, site_preferences_registry)
```

Here, we use django's built-in `AppConfig`, which is a convenient place to put this kind of logic.

To ensure this config is actually used by django, you'll also have to edit your app `__init__.py`:

```
# yourapp/__init__.py
default_app_config = 'yourapp.apps.YourAppConfig'
```

Warning: Ensure your app is listed **before** `dynamic_preferences` in `settings.INSTALLED_APPS`, otherwise, preferences will be collected before your registry is actually registered, and it will end up empty.

Start creating preferences

After this setup, you're good to go, and can start registering your preferences for the `Site` model in the same way you would do with the `User` model. You'll simply need to use your registry instead of the `user_preferences_registry`:

```
# yourapp/dynamic_preferences_registry.py
from dynamic_preferences.types import BooleanPreference, StringPreference, Section
from yourapp.registries import site_preferences_registry

access = Section('access')

@site_preferences_registry.register
class IsPublic(BooleanPreference):
    section = access
    name = 'is_public'
    default = False
```

Preferences will be available on your `Site` instances using the `preferences` attribute, as described in quickstart:

```
# somewhere in a view
from django.contrib.sites.models import Site

my_site = Site.objects.first()
if my_site.preferences['access__is_public']:
    print('This site is public')
```

Upgrade

Next

In order to fix #33 and to make the whole package lighter and more modular, user preferences where moved to a dedicated app.

If you were using user preferences before and want to use them after the package, upgrade will require a few changes to your existing code, as described below.

If you only use the package for the global preferences, no change should be required on your side, apart from running the migrations.

Add the app to your INSTALLED_APPS

In `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'dynamic_preferences',
    'dynamic_preferences.users.apps.UserPreferencesConfig',  # <---- add this line
]
```

Replace old imports

Some functions and classes were moved to the dedicated `dynamic_preferences.users` app.

The following imports will crash:

```
from dynamic_preferences.registry import user_preferences_registry
from dynamic_preferences.forms import (
    UserSinglePreferenceForm,
    user_preference_form_builder,
    UserPreferenceForm,
)
from dynamic_preferences.views import UserPreferenceFormView
from dynamic_preferences.models import UserPreferenceModel
```

You should use the following imports instead:

```
from dynamic_preferences.users.registry import user_preferences_registry
from dynamic_preferences.users.forms import (
    UserSinglePreferenceForm,
    user_preference_form_builder,
    UserPreferenceForm,
)
```

```
from dynamic_preferences.users.views import UserPreferenceFormView
from dynamic_preferences.users.models import UserPreferenceModel
```

Note: It is mandatory to update the path for `user_preferences_registry`. Other paths are part of the public API but their use is optional and varies depending of how you usage of the package.

Run the migrations

User preferences were stored on the `UserPreferenceModel` model class.

The migrations only rename the old table to match the fact that the modle was moved in another app. Otherwise, nothing should be deleted or altered at all, and you can inspect the two related migrations to see what we're doing:

- dynamic_preferences.0004_move_user_model
- dynamic_preferences.users.0001_initial

Anyway, please perform a backup before any database migration.

Once you're ready, just run:

```
python manage.py migrate dynamic_preferences_users
```

Note: If your own code was using ForeignKey fields pointing to `UserPreferenceModel`, it is likely your code will break with this migration, because your foreign keys will point to the old database table.

Such foreign keys were not officially supported or recommended though, and should not be needed in the uses cases dynamic_preferences was designed for. However, if you're in this situation, please file an issue on the issue tracker to see what we can do.

Remove useless setting

In previous versions, to partially adress #33, a `ENABLE_USER_PREFERENCES` setting was added to enable / disable the admin endpoints for user preferences. Since you can now opt into user preferences via `INSTALLED_APPS`, this setting is now obsolete and can be safely removed from your settings file.

0.8

Warning: there is a backward incompatbile change in this release.

To address #45 and #46, an import statement was removed from __init__.py. Because of that, every file containing the following:

```
from dynamic_preferences import user_preferences_registry, global_preferences_registry
```

Will raise an *ImportError*.

To fix this, you need to replace by this:

```
#                       .registries was added
from dynamic_preferences.registries import user_preferences_registry, global_preferences_registry
```

0.6

Sections are now plain python objects (see #19). When you use sections in your code, instead of the old notation:

```
from dynamic_preferences.types import BooleanPreference

class MyPref(BooleanPreference):
    section = 'misc'
    name = 'my_pref'
    default = False
```

You should do:

```
from dynamic_preferences.types import BooleanPreference, Section

misc = Section('misc')

class MyPref(BooleanPreference):
    section = misc
    name = 'my_pref'
    default = False
```

Note that the old notation is only deprecated and will continue to work for some time.

0.5

The 0.5 release implies a migration from `TextField` to `CharField` for `name` and `section` fields.

This migration is handled by the package for global and per-user preferences. However, if you created your own preference model, you'll have to generate the migration yourself.

You can do it via `python manage.py makemigrations <your_app>`

After that, just run a `python manage.py syncdb` and you'll be done.

Contributing

Important: We are using git-flow workflow here, so please submit your pull requests against develop branch (and not master).

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/EliotBerriot/django-dynamic-preferences/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

- Include whole stacktraces and error reports when necessary, directly in your issue body. Do not use external services such as pastebin.

Contributing

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

django-dynamic-preferences could always use more documentation, whether as part of the official django-dynamic-preferences docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/EliotBerriot/django-dynamic-preferences/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *django-dynamic-preferences* for local development.

1. Fork the *django-dynamic-preferences* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-dynamic-preferences.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-dynamic-preferences
$ cd django-dynamic-preferences/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 dynamic_preferences tests
$ python setup.py test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
    $ git add .
    $ git commit -m "Your detailed description of your changes."
    $ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python, 2.7, and 3.4. Check https://travis-ci.org/EliotBerriot/django-dynamic-preferences/pull_requests and make sure that the tests pass for all supported Python versions.
4. The pull request must target the *develop* branch, since the project relies on git-flow branching model

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_dynamic_preferences
```

Credits

Development Lead

- Eliot Berriot <contact@eliotberriot.com>

Contributors

- Ryan Anguiano, via prefs-n-perms package
- [willseward](https://github.com/willseward)
- [haroon-sheikh](https://github.com/haroon-sheikh)
- [yurtaev](https://github.com/yurtaev)
- [pomerama](https://github.com/pomerama)

- [philipbelesky](https://github.com/philipbelesky)
- [what-digital](https://github.com/what-digital)
- [czlee](https://github.com/czlee)
- [ricard33](https://github.com/ricard33)
- [JetUni](https://github.com/JetUni)
- [pip182](https://github.com/pip182)

Changelog

[Major release] 1.0 (21-02-2017)

Dynamic-preferences was release more than two years ago, and since then, more than 20 feature and bugfixe releases have been published. But even after two years the project was still advertised as in Alpha-state on PyPi, and the tags used for the releases, were implicitly saying that the project was not production-ready.

Today, we're changing that by releasing the first major version of dynamic-preferences, the `1.0` release. We will stick to semantic versioning and keep backward compatibility until the next major version.

Dynamic-preferences is already used in various production applications .The implemented features are stable, working, and address many of the uses cases the project was designed for:

- painless and efficient global configuration for your project
- painless and efficient per-user (or any other model) settings
- ease-of-use, both for end-user (via the admin interface) and developpers (settings are easy to create and to manage)
- more than decent performance, thanks to caching

By making a major release, we want to show that the project is trustworthy and, in the end, to attract new users and develop the community around it. Development will goes on as before, with an increased focus on stability and backward compatibility.

Because of the major version switch, some dirt was removed from the code, and manual intervention is required for the upgrade. Please have a look at https://django-dynamic-preferences.readthedocs.io/en/latest/upgrade.html for the detailed instructions.

Thanks to all the people who contributed over the years by reporting bugs, asking for new features, working on the documentation or on implementing solutions!

0.8.4 (10-01-2017)

This version is an emergency release to restore backward compatibility that was broken in 0.8.3, as described in issue #67. Please upgrade as soon as possible if you use 0.8.3.

Special thanks to [czlee](https://github.com/czlee) for reporting this!

0.8.3 (06-01-2017) (DO NOT USE: BACKWARD INCOMPATIBLE)

This release introduced by mistake a backward incompatible change (commit 723f2e). Please upgrade to 0.8.4 or higher to restore backward compatibility with earlier versions

This is a small bugfix release. Happy new year everyone!

- Now fetch model default value using the get_default method
- Fixed #50: now use real apps path for autodiscovering, should fix some strange error when using AppConfig and explicit AppConfig path in INSTALLED_APPS
- Fix #63: Added initial doc to explain how to bind preferences to arbitrary models (#65)
- Added test to ensure form submission works when no section filter is applied, see #53
- Example project now works with latest django versions
- Added missing max_length on example model
- Fixed a few typos in example project

0.8.2 (23-08-2016)

- Added django 1.10 compatibility [ricard33]
- Fixed tests for django 1.7
- Fix issue #57: PreferenceManager.get() returns value [ricard33]
- Fixed missing coma in boolean serializer [czlee]
- Added some documentations and example [JetUni]

0.8.1 (25-02-2016)

- Fixed still inconsistend preference order in form builder (#44) [czlee]

0.8 (23-02-2016)

Warning: there is a backward incompatbile change in this release. To address #45 and #46, an import statement was removed from __init__.py. Please refer to the documentation for upgrade instructions: http://django-dynamic-preferences.readthedocs.org/en/stable/upgrade.html

0.7.2 (23-02-2016)

- Fix #45: importerrror on pip install, and removed useless import
- Replaced built-in registries by persisting_theory, this will maintain a consistent order for preferences, see #44

0.7.1 (12-02-2016)

- Removed useless sections and fixed typos/structure in documentation, fix #39
- Added setting to disable user preferences admin, see #33
- Added setting to disable preference caching, fix #7
- Added validation agains sections and preferences names, fix #28, it could raise backward incompatible behaviour, since invalid names will stop execution by default

0.7 (12-01-2016)

- Added by_name and get_by_name methods on manager to retrieve preferences without using sections, fix #34
- Added float preference, fix #31 [philipbelesky]
- Made name, section read-only in django admin, fix #36 [what-digital]
- Fixed typos in documentation [philipbelesky]

0.6.6 (23-12-2015)

- Fixed #23 (again bis repetita): Fixed second migration to create section and name columns with correct length

0.6.5 (23-12-2015)

- Fixed #23 (again): Fixed initial migration to create section and name columns with correct length

0.6.4 (23-12-2015)

- Fixed #23: Added migration for shorter names and sections

0.6.3 (09-12-2015)

- Fixed #27: AttributeError: 'unicode' object has no attribute 'name' in preference *__repr__* [pomerama]

0.6.2 (24-11-2015)

- Added support for django 1.9, [yurtaev]
- Better travic CI conf (which run tests against two version of Python and three versions of django up to 1.9), fix #22 [yurtaev]

0.6.1 (6-11-2015)

- Added decimal field and serializer

0.6 (24-10-2015)

- Fixed #10 : added model choice preference
- Fixed #19 : Sections are now plain python objects, the string notation is now deprecated

0.5.4 (06-09-2015)

- Merged PR #16 that fix a typo in the code

0.5.3 (24-08-2015)

- Added switch for list_editable in admin and warning in documentation, fix #14
- Now use Textarea for LongStringPreference, fix #15

0.5.2 (22-07-2015)

- Fixed models not loaded error

0.5.1 (17-07-2015)

- Fixed pip install (#3), thanks @willseward
- It's now easier to override preference form field attributes on a preference (please refer to Preferences attributes for more information)
- Cleaner serializer api

0.5 (12-07-2015)

This release may involves some specific upgrade steps, please refer to the `Upgrade` section of the documentation.

0.5 (12-07-2015)

This release may involves some specific upgrade steps, please refer to the `Upgrade` section of the documentation.

- Migration to CharField for section and name fields. This fix MySQL compatibility issue #2
- Updated example project to the 0.4 API

0.4.2 (05-07-2015)

- Minor changes to README / docs

0.4.1 (05-07-2015)

- The cookiecutter part was not fully merged

0.4 (05-07-2015)

- Implemented cache to avoid database queries when possible, which should result in huge performance improvements
- Whole API cleanup, we now use dict-like objects to get preferences values, which simplifies the code a lot (Thanks to Ryan Anguiano)
- Migrated the whole app to cookiecutter-djangopackage layout
- Docs update to reflect the new API

0.3.1 (10-06-2015)

- Improved test setup
- More precise data in setup.py classifiers

0.2.4 (14-10-2014)

- Added Python 3.4 compatibility

0.2.3 (22-08-2014)

- Added LongStringPreference

0.2.2 (21-08-2014)

- Removed view that added global and user preferences to context. They are now replaced by template context processors

0.2.1 (09-07-2014)

- Switched from GPLv3 to BSD license

P