django-usersettings Documentation

Release 0.1

Alexander van Ratingen

December 04, 2012

CONTENTS

This documentation covers the 0.1 release of django-usersettings, a simple pluggable application to manage user settings for Django-powered websites.

Your apps can define and register user settings: configuration users can manage to specify how they'd like your site to work. Helpers are provided for standard setting patterns, functions for changing settings and form factories to serve your users with.

To get up and running quickly, consult the *quick-start guide*, which describes all the necessary steps to install django-usersettings and start using user settings. For more detailed information read through the documentation listed below.

Contents:

CHAPTER

ONE

QUICK START GUIDE

As this is an app for websites built on the Django framework, you'll need to have that installed. The minimum required versions are Python 2.5 (no python 3 support yet) and Django 1.2. To store the settings, you'll need a database. Any database that's supported by Django ORM will do.

1.1 Installing django-usersettings

There are several ways to install django-usersettings:

- Automatically, via a package manager.
- Manually, by downloading a copy of the release package and installing it yourself.
- Manually, by performing a Mercurial checkout of the latest code.

It is also highly recommended that you learn to use virtualenv for development and deployment of Python software; `virtualenv` provides isolated Python environments into which collections of software (e.g., a copy of Django, and the necessary settings and applications for deploying a site) can be installed, without conflicting with other installed software. This makes installation, testing, management and deployment far simpler than traditional site-wide installation of Python packages.

1.1.1 Automatic installation via a package manager

Several automatic package-installation tools are available for Python; the most popular are easy_install and pip. Either can be used to install django-usersettings.

Using `easy_install`, type:

```
easy_install -Z django-usersettings
```

Note that the `-Z` flag is required, to tell `easy_install` not to create a zipped package; zipped packages prevent certain features of Django from working properly.

Using `pip`, type:

```
pip install django-usersettings
```

It is also possible that your operating system distributor provides a packaged version of django-usersettings. Consult your operating system's package list for details, but be aware that third-party distributions may be providing older versions of django-usersettings, and so you should consult the documentation which comes with your operating system's package.

1.1.2 Manual installation from a downloaded package

If you prefer not to use an automated package installer, you can download a copy of django-usersettings and install it manually. The latest release package can be downloaded from django-usersettings's listing on the Python Package Index.

Once you've downloaded the package, unpack it (on most operating systems, simply double-click; alternately, type `tar zxvf django-usersettings-0.1.tar.gz` at a command line on Linux, Mac OS X or other Unix-like systems). This will create the directory `django-usersettings-0.1`, which contains the `setup.py` installation script. From a command line in that directory, type:

```
python setup.py install
```

Note that on some systems you may need to execute this with administrative privileges (e.g., `sudo python setup.py install`).

1.1.3 Manual installation from a Mercurial checkout

If you'd like to try out the latest in-development code, you can obtain it from the django-usersettings repository, which is hosted at Bitbucket and uses Mercurial for version control. To obtain the latest code and documentation, you'll need to have Mercurial installed, at which point you can type:

```
hg clone http://bitbucket.org/Blue/django-usersettings/
```

You can also obtain a copy of a particular release of django-usersettings by specifying the `-r` argument to `hg clone`; each release is given a tag of the form `vX.Y`, where "X.Y" is the release number. So, for example, to check out a copy of the 0.1 release, type:

```
hg clone -r v0.1 http://bitbucket.org/Blue/django-usersettings/
```

In either case, this will create a copy of the django-usersettings Mercurial repository on your computer; you can then add the `django-usersettings` directory inside the checkout your Python import path, or use the `setup.py` script to install as a package.

1.2 Basic configuration and use

Once installed, you can add django-usersettings to any Django-based project you're developing. Begin by adding `usersettings` to the `INSTALLED_APPS` setting of your project. This is enough to get started. For more control, there are a few settings to customize things.

`USERSETTINGS_DATABASE_ALIAS` This optional settings can be set to the name of a database alias to store the settings in. The default is to store settings in a the default database.

`USERSETTINGS_GROUPING_SEPARATOR` A character (or string, if you'd want) that marks the grouping elements of a setting grouping string. This defaults to a colon (`:`), resulting in grouping strings like `formatting:datetime:dates`. Set it to whatever you want if you happen to find that notation perverse. These groupings can be used to *filter loading settings*.

`USERSETTINGS_CACHE_ALIAS` Set this to the alias of a cache to speed up setting retrieval. Omitting this, or setting it to `None` will deactivate caching settings.

`USERSETTINGS_CACHE_KEY_FORMAT` When caching settings, this is the format for the keys used to store cached settings. This uses the new string formatting available since python 2.6. The arguments passed are `user` and `name`. The default value used is `usersettings_{user.id}_{name}`. You should always include both a unique field of the user and the setting name to ensure that the keys are one-to-one mappings to user settings.

1.2.1 Setting up URLs

The app includes a Django `URLconf` which sets up URL patterns for *the views in django-usersettings*. This `URLconf` can be found at `usersettings.urls`, and so can simply be included in your project's root URL configuration. For example, to place the URLs under the prefix `/settings/`, you could add the following to your project's root `URLconf`:

```
(r'^settings/', include('usersettings.urls')),
```

This would then be the index page for managing settings. To completely customize the url locations, add the urlpatterns for the *included views* yourself. If you go down this road, do make sure that the url names are still the same.

1.2.2 Templates

When you use the builtin views, and you don't specify custom locations for the templates (like when including the builtin `usersettings.urls` patterns as described above) make sure you create the following templates.

Note that all of these are rendered using a `RequestContext` and so will also receive any additional variables provided by context processors.

usersettings/form.html

Used to show the form users will update their settings with.

`form` The form instance to display to the user.

1.2.3 Using settings

Only thing that rests is defining settings and using them. Settings are defined by subclassing a base class called `usersettings.Setting`. Three method must be implemented; one that gives the default value for a setting and two others that convert a setting object (anything you want it to be) into a string to save to the database. Additionally, you'll have to specify a name for the setting. This will be the name to retrieve the setting by. The `verbose_name` and `description` attributes are optional, they are used in the builtin views to generate pretty forms.

```
from django.utils.translation import ugettext_lazy as _
from usersettings import Setting, register

class WelcomeMessageSetting(Setting):
    name = 'welcome_message'
    verbose_name = _("welcome message")
    description = _("The message you'd like to see when you enter the website.")

    def encode(self, user, setting):
        "Turn the setting object in a string to store in the database."
        return setting

    def decode(self, user, value):
        "Turn the stored string into the setting object."
        return value

    def default(self, user):
        "The default setting object."
        return 'Welcome {}'.format(user)

register.register(WelcomeMessageSetting)
```

To allow this to be used in automatically generated forms or the included views, you should also implement two additional methods. The formfield you generate does not need to have a `label` or a `help_text` if you specified the `verbose_name` and `description` attributes on the `Setting` subclass.

```
from django import forms
from usersettings import Setting

class WelcomeMessageSetting(Setting):

    # ...

    def form_field(self, user):
        "Get a dict of formfields to change this setting."
        return forms.CharField()

    def encode_form_data(self, user, data):
        "Turn the form data into the string to store in the database."
        return self.encode(user, data)
```

If your setting requires several formfields to specify, use Django's `MultiValueField`.

This example was intentionally very simplictic. Since the setting object is a string, the decoding and encoding does not require any work.

If you want to use custom python objects as setting values, you might like to use the `PickleSetting` subclass of `Setting` since it does not have the requirement that the raw database stored object that decode and encode use, is a string. To illustrate that the setting object can be any python object, the following example is a trimmed version of the included `usersettings.contrib.FilesizeFormatSetting`:

```
class FilesizeFormatSetting(PickleSetting):

    def encode(self, user, setting):
        if isinstance(setting, tuple) and len(setting) == 2:
            return setting
        else:
            raise ValueError

    def decode(self, user, value):
        def filesize_formatter(filesize):
            if filesize == 0:
                return '0 B'
            base = 1024 if value[1] else 1000
            # compute, and set result to the format
            return format
        return filesize_formatter

    def default(self, user):
        return (1024, True)

    def form_field(self, user):
        return MultiValueField(fields=[
            IntegerField(label=''),
            BooleanField(label='binairy prefixes', blank=True),
        ])
```

CHAPTER

TWO

RELEASES

2.1 usersettings 0.1

The initial release

CHAPTER

THREE

DEFINING SETTINGS

CHAPTER

FOUR

USING SETTINGS

CHAPTER

FIVE

GENERATING FORMS

CHAPTER

SIX

VIEWS

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS

These are some of the questions I've gotten about this project.

`This is pretty awesome` Ok, this is not really a question, but let me give it a go. Yes, it most certainly is!

See Also:

- The source code, to find out about django-usersettings internals.