django-signup Documentation

Release 0.6.0

Felipe Bessa Coelho

Sep 15, 2017

Contents

`django-signup` is a Django app that handles user registration.

Contents:

CHAPTER 1

Installation

To install `django-signup`, use pip:

```
pip install django-signup
```

If you want the latest-and-greatest version, install it with

```
pip install -e  hg+https://bitbucket.org/fcoelho/django-signup#egg=django-signup
```

CHAPTER 2

Setup

You have to add the app to `INSTALLED_APPS` and also include some URLs in your `urlconf`:

```
INSTALLED_APPS = (
    ...
    'signup',
    ...
)
```

```
urlpatterns = patterns('',
    ...
    url(r'^accounts/', include('signup.urls')),
    ...
)
```

Settings

These are the settings specific to `django-signup`.

SIGNUP_ACTIVATION_DAYS

This controls for how many days an instance of `signup.models.Validation` will be valid before being considered "expired". It is used on the *clear_expired_signups* management command.

If not set in the user settings, the default value is **2 days**.

SIGNUP_FORM_CLASS

A string with the full dotted name of a class representing the form to be used on the signup page.

If not specified, the `signup.forms.DefaultUserCreationForm` class is used.

SIGNUP_ALLOWED

A boolean indicating whether signup is allowed. If `True`, nothing special happens. If `False`, all requests to the account creation page will be redirected to a template saying that registration is closed.

New in version 0.5.

This can also be a dotted path to a callable that takes no arguments and returns a boolean indicating whether signup is allowed or not.

Signals

New in version 0.6.

There are two signals defined in the `signup.signals` module: `user_signed_up` and `user_activated`. The first is sent right after the user object has been created. The second is sent after the user has hit his activation URL and the corresponding `Validation` object has been deleted.

Both signals are sent with two arguments: `user`, the user object, and `request`, the view request object.

CHAPTER 3

Templates

The app does not ship with any templates. There are some templates in the sample project available in the source distribution, but those are **extremely simple and for demonstration purposes only**.

These are the templates you should create at the very least:

registration/signup_form.html
: Holds the registration form. It is displayed by a view which inherits from FormView. It has the following context:

  - `form`: an instance of the configured `SIGNUP_FORM_CLASS` or an instance of `django.contrib.auth.forms.UserCreationForm`

registration/signup_closed.html
: A simple page saying that registration is closed. It has no custom context.

registration/signup_complete.html
: A simple page indicating that the account has been created and an email has been sent to finish the sign up process. It has no special context.

registration/activation_failed.html
: Should display a message indicating that the activation failed. It has the following context:

  - `activation_key`: The key used to activate the account. It's part of the URL

registration/activation_complete.html
: Should display a message saying that the account was successfully activated. It has no special context.

registration/activation_email.txt
: Contains the email subject and body, in text and optional html format. This template has a special "syntax" in order to include the subject and body in the same file. A sample file is shown below. Empty lines at the begin and end of each block are stripped away.

```
-- title=subject

This is the subject of the email,

multiple lines will be joined
```

```
-- title=txt
This is the text part of the email

-- title=html
This is the html part of the email, it's optional
```

This template has the following context:

- `url`: The full url to be used to activate the account. Includes the domain and path
- `activation_days`: How many days the user has to activate his account

CHAPTER 4

Management commands

Until now, there is only one management command, documented below.

clear_expired_signups

This command is used to delete from the database user registrations that are more than *SIGNUP_ACTIVATION_DAYS* days old and haven't been activated yet.

It deletes both the `signup.models.Validation` instance and the user model instance. By that time, it's worth noting that the username[1] registered by the original user becomes available again. If the original user still wants to keep that username, it has to register again.

[1] "username", here, refers to the field marked as USERNAME_FIELD in current installed user model.

CHAPTER 5

Admin interface

`django-signup` adds its `signup.models.Validation` class to the admin interface for easy inspection of the current validations.

In addition, you can also resend the activation emails if necessary. To do that, select the target validation instances and, from the drop down menu, select `Resend activation email`.

CHAPTER 6

Using custom user models

This app was created to be used with custom user models, which were introduced in Django 1.5. While the basic configuration in *Setup* allows you to use `django-signup` with the default user model, some extra configuration is needed in order to work with custom user models.

The custom user model

Consider the following user model and associated manager:

```
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None):
        if not email:
            raise ValueError('missing email')
        user = self.model(email=email)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password):
        #for this example, nothing special happens here
        return self.create_user(email, password)

class CustomUser(AbstractBaseUser):
    email = models.EmailField(
        max_length=254,
        unique=True,
        db_index=True
    )

    objects = CustomUserManager()
```

```
    USERNAME_FIELD = 'email'

    def get_short_name(self):
        return self.email
    def get_full_name(self):
        return self.email
```

This model has, in practice, two fields: `email` and `password`, the latter which is provided by `AbstractBaseUser`.

The custom sign up form

Requirements

When this user model is installed (see the docs), it is already available to Django, but `django-signup` requires you to create a custom version of what Django provides as UserCreationForm.

This form should have all the necessary information needed to create a custom user model instance through the `CustomUser.objects.create_user` method. What that really means is:

- For every field in `CustomUser.REQUIRED_FIELDS` plus `CustomUser.USERNAME_FIELD`, there should be a field of the same name in the sign up form;
- The only exception is the `password` field, which is populated from three possible different sources: `password`, `password1`, or `password2`. This is to accommodate forms which have duplicated fields for password checking (which you should do anyway).

  When calling `create_user`, `django-signup` will use the first available value from those three options, so make sure that, if you have more than one password field, the values match. Ensure that with a `clean_*` method on the form, as shown below.

  The drawback of this approach is that you have to call your password fields `password{,1,2}` instead of, say, `pw` or anything else.

Implementation

For the user model described above, this form might look like the following:

```
from django import forms
from your_app.models import CustomUser

class UserSignUpForm(forms.Form):
    email = forms.EmailField()
    password1 = forms.CharField(widget=forms.PasswordInput)
    password2 = forms.CharField(widget=forms.PasswordInput)

    def clean_email(self):
        email = self.cleaned_data['email'].strip()
        try:
            CustomUser.objects.get(email__iexact=email)
            raise forms.ValidationError('email already exists')
        except CustomUser.DoesNotExist:
            return email

    def clean_password2(self):
```

```
        pw1 = self.cleaned_data.get('password1')
        pw2 = self.cleaned_data.get('password2')
        if pw1 and pw2 and pw1 == pw2:
            return pw2
        raise forms.ValidationError("passwords don't match")
```

Note that you should implement whatever logic you need to verify your data here. In this case, we're checking for uniqueness of the email field and that both passwords match. The email field is also going to be checked when the new user is about to be inserted at the database, but by performing our check in the form we're able to provide the user with a meaningful message about why the signup process didn't go as expected.

Using the new form

Next, you have to tell `django-signup` that you want to use this specific form during the registration process. Suppose that the above form is inside the `your_app/forms.py` file. Then, you have to add the following to your project settings:

```
SIGNUP_FORM_CLASS = 'your_app.forms.UserSignUpForm'
```

CHAPTER 7

Changelog

0.6.0

- Added the `user_signed_up` and `user_activated` signals

0.5.0

- Allows `SIGNUP_ALLOWED` to be a path to a callable so you can decide at runtime whether signup is allowed or not

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

R

T