Django-le-social Documentation

Release 0.9

Bruno Renié

October 17, 2016

Contents

Django-le-social is an external registration helper for Django. It currently lets you use Twitter (OAuth) and OpenID authentication, as well as traditional registration.

CHAPTER 1

Design

It’s more a framework than a drop-in app in the sense that it won’t create any user data for you: when a user comes from an external authentication source, django-le-social executes a method that **you** decide. There is no user creation, no new model instance, no user login. You need to decide what to do, mainly store the OAuth token or the OpenID data, create a user and log him in.

The model structure is completely up to you: you can use django-le-social with any user model or session backend, you should be able to plug it to almost any existing project. If you can’t, it’s probably a bug – please report back!

Django-le-social **doesn’t add any settings**. While you can store some stuff in the settings, it’s not enforced. For application logic, we try to use attributes and methods as much as possible.

Code

The source code is available on Github under the 3-clause BSD license.

CHAPTER 3

Installation

If you have Django 1.8 and above:

```
pip install django-le-social
```

If you have Django 1.4 - 1.7:

```
pip install "django-le-social<0.9"
```

Django-le-social is tested for python 2.7, 3.4 and 3.5.

CHAPTER 4

Usage

4.1 Twitter authentication

4.1.1 Requirements

Install the twitter library:

```
pip install twitter<1.8
```

4.1.2 Basic usage

Communications with Twitter are handled with Mike Verdone's minimalist python twitter API library. Define two URLs, one to initiate the twitter login and the other for the OAuth callback:

```
from myapp import views

urlpatterns = patterns('',
    url(r'^oauth/authorize/$',
        views.authorize,
        name='oauth_authorize'),
    # use the following URL if you want to force authentication
    # For example, if you're already authenticated, but want to
    # reauthenticate as a different user.
    url(r'^oauth/authorize/force/$',
        views.authorize,
        {'force_login': True},
        name='oauth_force_authorize'),
    url(r'^oauth/callback/$',
        views.callback,
        name='oauth_callback'),
)
```

Set your OAuth consumer key and secret in your settings:

```
CONSUMER_KEY = 'yayyaaa'
CONSUMER_SECRET = 'whoooooohooo'
```

And create the two views:

```
from django.http import HttpResponse
from django.shortcuts import redirect
from django.utils import simplejson as json

import twitter

from le_social.twitter import views

authorize = views.Authorize.as_view()

class Callback(views.Callback):
    def error(self, message, exception=None):
        return HttpResponse(message)

    def success(self, auth):
        api = twitter.Twitter(auth=auth)
        user = api.account.verify_credentials()
        dbuser, created = SomeModel.objects.get_or_create(
            screen_name=user['screen_name']
        )
        user.token = auth.token
        user.token_secret = auth.token_secret
        user.save()
        return redirect(reverse('some_view'))
callback = Callback.as_view()
```

On the `Callback` view, you need to implement the `error(message, exception=None)` and `success(auth)` methods. Both must return an HTTP response.

4.1.3 Extension points

Authorize

The `Authorize` is a `django.views.generic.View` subclass. Customization can be done using the extension points it provides. For instance, if one doesn't want to allow logged-in users to sign in with Twitter:

```
class Authorize(views.Authorize):
    def get(self, request, *args, **kwargs):
        if request.user.is_authenticated():
            return redirect('/')
        return super(Authorize, self).get(request, *args, **kwargs)
authorize = Authorize.as_view()
```

If you want Twitter to redirect your user to a custom location, specify it in `Authorize.build_callback`. This function needs to return an absolute URI, including protocol and domain. For instance:

```
from django.contrib.sites.models import Site

# We're replacing the following line:
# authorize = views.Authorize.as_view()

class Authorize(views.Authorize):
    def build_callback(self):
        # build a custom callback URI
        next = self.request.path
        site = Site.objects.get_current()
        return 'http://{0}{1}?next={2}'.format(
            site.domain,
            reverse('oauth_callback'),
            next)
```

If you don't implement `build_callback` or if you return `None`, your users will be redirected to the default URL specified in the app's settings on twitter.com.

Although you can specify a default, it is good practice to always pass a callback URI when authorizing; this is the preferred way to preserve application state when the user's browser returns from authenticating.

Don't forget to update your urlconf after defining a custom callback URL. Returning browsers should be routed to the Callback view.

Callback

You can also special-case the `Callback` view using the same technique, but you really need to implement the `error()` and `success()` methods on this class.

OAuth credentials

By default, the `Authorize` and `Callback` views look for the Twitter app credentials in your settings (`CONSUMER_KEY`, `CONSUMER_SECRET`). You can implement your own mixin instead. The default OAuth mixin looks for the consumer key and secrets in this order:

- `consumer_key` and `consumer_secret` as attributes on the view class,
- `settings.CONSUMER_KEY` and `settings.CONSUMER_SECRET`

If you set `consumer_key` and `consumer_secret` on the class, you need to do so on the two views, or make your custom views inherit from a mixin that provides them.

For more logic, you can also re-implement `get_consumer_key()` and `get_consumer_secret()` on the view classes to use different consumers under certain conditions:

```
class OAuthMixin(views.OAuthMixin):
    def get_consumer_key(self):
        if self.request.user.username == 'bruno':
            return 'hahahah'
        return super(OAuthMixin, self).get_consumer_key()

class Authorize(OAuthMixin, views.Authorize):
    pass
authorize = Authorize.as_view()

class Callback(OAuthMixin, views.Callback):
    def success(self, auth):
        do_some_stuff()
        return something
callback = Callback.as_view()
```

4.2 OpenID authentication

4.2.1 Requirements

Install the `python-openid` package:

```
pip install python-openid
```

For OpenID support, you need `le_social.openid` in your `INSTALLED_APPS`. Make sure you run `manage.py syncdb`.

If you want to access the list of OpenID URLs associated to the current session, add `le_social.middleware.OpenIDMiddleware` to your `MIDDLEWARE_CLASSES`. This will add an `openids` attribute to incoming requests. `request.opendis` is a list of `le_social.openid.utils.OpenID` objects.

4.2.2 Basic usage

Define the two URLs to initiate the OpenID connection and the return URL:

```
from myapp import views

urlpatterns = patterns('',
    url(r'^openid/$', views.begin, name='openid_begin'),
    url(r'^openid/complete/$', views.callback, name='openid_complete'),
)
```

And define your two view using the base classes provided by django-le-social:

```
from django.http import HttpResponse

from le_social.openid import views

class Begin(views.Begin):
    return_url = '/openid/complete/'
    template_name = 'openid.html'

    def failure(self, message):
        return HttpResponse(message)
begin = Begin.as_view()

class Callback(views.Callback):
    return_url = '/openid/complete/'

    def success(self):
        openid_url = self.openid_response.identity_url
        # self.openid_response contains the openid info
        return HttpResponse('Openid association: %s' % openid_url)

    def failure(self, message):
        return HttpResponse(message)
callback = Callback.as_view()
```

You also need a basic template to render the OpenID form:

```
<form method="post" action=".">
    {{ form.as_p }}
    {% csrf_token %}
    <input type="submit" value="Sign in">
</form>
```

This code will just return the OpenID URL in case of successful authentication. Usually in the `success()` method, you would need to store the OpenID URL in the DB, attach it to the currently logged-in user or create a new user object.

The `failure()` methods are here to handle authentication failures, when the OpenID URL isn't valid or something goes wrong during the OpenID negociation.

4.2.3 Extension points

Return URL

Both the `Begin` and `Callback` views need a `return_url` attribute. In the examples above the URL is hardcoded but you can provide a dynamically-generated one by defining `get_return_url()` on the view class or on a mixin shared by your subclasses:

```python
from django.core.urlresolvers import reverse

from le_social.openid import views

class ReturnUrlMixin(object):
    def get_return_url(self):
        return reverse('openid_complete')

class Begin(ReturnUrlMixin, views.Begin):
    pass
begin = Begin.as_view()

class Callback(ReturnUrlMixin, views.Callback):
    def success(self):
        return something
callback = Callback.as_view()
```

Form class

The `Begin` view is a standard `FormView` that takes a `form_class` attribute. The default value is `le_social.openid.forms.OpenIDForm`, it just asks for a valid URL. If you want to do more specific validation, subclass the form and override `clean_openid_url()`.

Sreg attributes

The `sreg_attrs` dictionnnary on the `Begin` class defines which Sreg fields to ask for. By default it is an empty dict but if you don't specify anything it automatically gets updated to `{'optional': ['nickname', 'email']}`.

You can alter the `sreg_attrs` attribute or implement `get_sreg_attrs()` on the view class.

Attribute Exchange

The `ax_attrs` attribute on the `Begin` class defines which AX attributes to request. By default it is an empty list. If you need to set this dynamically, implement `get_ax_attrs()`.

Trust Root

By default the trust root is the root of your website. If you want to change it, alter the `trust_root` attribute on the `Begin` class, or define `get_trust_root()`. Note that `trust_root` must be a URL without the host (e.g. `'/something/'`), whereas `get_trust_root()` must return a full URL, including the protocol and host name.

4.2.4 OpenID objects

With the `OpenIDMiddleware`, the request gets an `openids` attribute, a list of the OpenIDs associated to the current session. Each element is a `le_social.openid.utils.OpenID` instance and has the following information attached:

- `openid`: the OpenID URL
- `issued`: the time when the association was successful
- `attrs`: the OpenID attributes
- `sreg`: the Sreg attributes
- `ax`: the AX attributes.

4.3 Traditional Registration

Note: Django versions

Django-le-social 0.6 requires Django 1.4 or greater. If you still run Django <= 1.3, use django-le-social==0.5.

This part explains how to use le-social to handle *traditional* registration, ala django-registration.

Here's the workflow:

- A user visits your site
- He clicks "register"
- He fills a form asking him for some details
- He gets a notification (email, SMS, postcard, rocket) with a secret activation link
- He follows the link and his account is activated

You need to add to your project:

- The URLs.
- If you need something different than the default scenario, an implementation of the registration and activation logic.

Everything you need is under the `le_social.registration` namespace.

4.3.1 Basic Usage

This example will show you how to implement the equivalent of django-registration.

Note: Templates

No templates are provided with django-le-social. See the end of this page for the default template paths.

First, create an app. Let's call it `registration`:

```
python manage.py startapp registration
```

Add some URLs in `registration/urls.py`:

```
from django.conf.urls import patterns, url

from . import views

urlpatterns = patterns('',
    url(r'^activate/complete/$', views.activation_complete,
        name='registration_activation_complete'),

    url(r'^activate/(?P<activation_key>[^/]+)/$', views.activate,
        name='registration_activate'),

    url(r'^register/$', views.register,
        name='registration_register'),

    url(r'^register/complete/$', views.registration_complete,
        name='registration_complete'),

    url(r'^register/closed/$', views.registration_closed,
        name='registration_closed'),
)
```

Finally, add the `registration.views` you referenced in `urls.py`. In this example, we'll be using the default behaviour that creates an inactive Django user on registration, sends him a verification email and activates his account when he clicks on the activation link.

```
from le_social.registration import views

register = views.Register.as_view()
registration_complete = views.RegistrationComplete.as_view()
registration_closed = views.RegistrationClosed.as_view()

activate = views.Activate.as_view()
activation_complete = views.ActivationComplete.as_view()
```

4.3.2 Extension points

Registration form

`le_social.registration.views.Register` is a FormView. The default registration form asks for:

- A username
- An email address
- Two passords

The default form only checks that the email is correct and the two passwords match. If you want to perform extra validation, such as checking that the username and the email are unique, just subclass the form and add your validation logic:

```
from django import forms
from le_social.registration.forms import RegistrationForm

class MyRegistrationForm(RegistrationForm):
    def clean_username(self):
        if User.objects.filter(
            username=self.cleaned_data['username'],
        ).exists():
            raise forms.ValidationError('This username is already being used')
        return self.cleaned_data['username']
```

Then declare your custom form in the `Register` view. Instead of doing:

```
register = views.Register.as_view()
```

Do:

```
from .forms import MyRegistrationForm

register = views.Register.as_view(
    form_class=MyRegistrationForm,
)
```

Or even:

```
from .forms import MyRegistrationForm

class Register(views.Register):
    form_class = MyRegistrationForm
register = Register.as_view()
```

You can also completely rewrite the registration form to ask for different fields. However, there are a couple of requirements for this form:

- It **must** implement a `save()` method. The default form's `save()` implementation inserts a new `User` object from `django.contrib.auth`. If you need a custom user model, define `save()` on your form to create a different object.
- The `save()` method **must** return a `User` object, or any model instance that has a primary key. This object is added to the template context for the registration notification (see below) and the primary key is used to generate the activation link.

Registration notification

The `Register` view has a `send_notification()` method that sends an activation email by default. The following templates are used:

- `le_social/registration/activation_email.txt` for the email body,
- `le_social/registration/activation_email_subject.txt` for the email subject.

The following context variables are available:

- `user`: the `User` instance returned by your form's `save()` method.
- `site`: a `RequestSite` object from the current request.
- `activation_key`: the signed key to put in your activation link. You can build the activation link like this:

```
    http://{{ site.domain }}{% url "registration_activate" activation_key %}
```

If you need more context variables, override `get_notification_context()` on the `Register` view. For instance, to add a `scheme` variable containing either `http` or `https`:

```
class Register(views.Register):
    def get_notification_context(self):
        context = super(Register, self).get_notification_context()
        context.update({
            'scheme': 'https' if self.request.is_secure() else 'http'
        })
        return context
```

Other registration parameters

The following attributes of the `Register` class can be customized:

- `closed_url`: the URL to redirect to if the registration is closed. Defaults to `reverse('registration_closed')`.
- `form_class`: the form to use for registration. Defaults to `le_social.registation.forms.RegistrationForm`.
- `registration_closed`: boolean to open or close the registration. Defaults to `False`.
- `success_url`: the URL to redirect to on successful registration. Defaults to `reverse('registration_complete')`.
- `template_name`: the template to use to render the registration form. Defaults to `'le_social/registration/register.html'`.
- `notification_template_name`: the template to use for the notification email. Defaults to `'le_social/registration/activation_email.txt'`.
- `notification_subject_template_name`: the template to use for the notification subject. Defaults to `'le_social/registration/activation_email_subject.txt'`.

The following methods can be customized:

- `get_registration_closed()`: returns the value of `registration_closed`.
- `get_closed_url()`: returns the value of `closed_url`.
- `get_notification_context()`: builds the template context for the activation email.
- `send_notification()`: sends the activation notification. This is an email by default, but you can override this method to do anything else instead.

Activation view

The `Activate` view is a simple `TemplateView` that loads the activation key into an `activation_key` attribute.

The key is signed using your `SECRET_KEY` setting. If the key is properly loaded, the activation view calls the `activate()` method and redirects to a `get_success_url()`.

If the key is not valid, the template is rendered. Hence the template should show a “unable to activate” message, or something similar.

The following attributes can be set on the `Activate` view:

- `template_name`: the template to use in case of failed activation. Defaults to `'le_social/registration/activate.html'`.
- `success_url`: the URL to redirect to in case of successful activation. Defaults to `reverse('registration_activation_complete')`.
- `expires_in`: the delay (in seconds) after which an activation link should be considered as expired. Defaults to `2592000` (30 days), set it to `None` if you want them to never expire.

The following methods can be overriden:

- `get_expires_in()`: returns the content of `expires_in` by default.
- `get_success_url()`: returns the content of `success_url`.

- `activate()`: sets the user's `is_active` attribute to `True`. Override it if you have a custom user model.

4.3.3 Other registration views

The other views are plain `TemplateViews`, their templates are not provided either. Here are the default paths, which you can alter using `template_name`.

- `RegistrationComplete`: renders `le_social/registration/registration_complete.html`.
- `RegistrationClosed`: renders `le_social/registration/registration_closed.html`.
- `ActivationComplete`: renders `le_social/registration/activation_complete.html`.

CHAPTER 5

Changes

- 0.8:
  - The `activate()` method of `le_social.registration.views.Activate` now has access to the request parameters in `self.request`, `self.args` and `self.kwargs`. This is useful if you need the request object for automatically logging the user in for instance.
  - Custom user model support in `le_social.registration`. If your user model is created differently than with a username, an email and a password you need to override the registration form. Furthermore if your user doesn't have an `is_active` flag you need to override the `activate()` method of the activation view.
- 0.7:
  - Fixes for Django 1.5 and Python 3.
- 0.6:
  - Travis tests
  - Django requirement bumped to 1.4
  - `itsdangerous` requirement dropped
  - `twitter` and `python-openid` requirements made optional: if you only use `le_social.registration` you don't need to install them.
- 0.5:
  - Tox tests for python 2.6 / 2,7 and Django 1.2 / 1.3 / trunk.
  - Changed the registration API, backwards-incompatible if you were using it but *much* simpler to use.
- 0.4:
  - Test suite
  - Django < 1.3 compatibility with django-cbv
  - "Traditional" registration support, ala django-registration
- 0.3:
  - switched from tweepy to twitter for Twitter authentication
  - added the ability to force the login on the twitter authorization screen
- 0.2:
  - renamed OpenID's and Twitter's `Return` views to `Callback`

    - added `build_callback` for custom twitter callback URLs

- 0.1: initial version

CHAPTER 6

Indices and tables

- genindex
- modindex
- search