Python Social Auth Documentation

Release

Matías Aguirre

Nov 03, 2017

Contents

Python Social Auth aims to be an easy to setup social authentication and authorization mechanism for Python projects supporting protocols like OAuth (1 and 2), OpenId and others.

The initial codebase is derived from django-social-auth with the idea of generalizing the process to suit the different frameworks around, providing the needed tools to bring support to new frameworks.

django-social-auth itself was a product of modified code from django-twitter-oauth and django-openid-auth projects.

The project is now split into smaller modules to isolate and reduce responsibilities and improve reusability.

Contents:

CHAPTER 1

Introduction

Python Social Auth aims to be an easy to setup social authentication and authorization mechanism for Python projects supporting protocols like OAuth (1 and 2), OpenId and others.

1.1 Features

This application provides user registration and login using social sites credentials, here are some features, probably not a full list yet.

1.1.1 Supported frameworks

Multiple frameworks support:

- Django
- Flask
- Pyramid
- Webpy
- Tornado

More frameworks can be added easily (and should be even easier in the future once the code matures).

1.1.2 Auth providers

Several supported service by simple backends definition (easy to add new ones or extend current one):

- Angel OAuth2
- Beats OAuth2
- Behance OAuth2

- Bitbucket OAuth1
- Box OAuth2
- Dailymotion OAuth2
- Deezer OAuth2
- Disqus OAuth2
- Douban OAuth1 and OAuth2
- Dropbox OAuth1
- Evernote OAuth1
- Facebook OAuth2 and OAuth2 for Applications
- Fitbit OAuth2 and OAuth1
- Flickr OAuth1
- Foursquare OAuth2
- Google App Engine Auth
- Github OAuth2
- Google OAuth1, OAuth2 and OpenId
- Instagram OAuth2
- Kakao OAuth2
- Linkedin OAuth1
- Live OAuth2
- Livejournal OpenId
- Mailru OAuth2
- MineID OAuth2
- Mixcloud OAuth2
- Mozilla Persona
- NaszaKlasa OAuth2
- NGPVAN ActionID OpenId
- Odnoklassniki OAuth2 and Application Auth
- OpenId
- Podio OAuth2
- Pinterest OAuth2
- Rdio OAuth1 and OAuth2
- Readability OAuth1
- Shopify OAuth2
- Skyrock OAuth1
- Soundcloud OAuth2
- Spotify OAuth2

- ThisIsMyJam OAuth1
- Stackoverflow OAuth2
- Steam OpenId
- Stocktwits OAuth2
- Stripe OAuth2
- Tripit OAuth1
- Tumblr OAuth1
- Twilio Auth
- Twitch OAuth2
- Twitter OAuth1
- Upwork OAuth1
- Vimeo OAuth1
- VK.com OpenAPI, OAuth2 and OAuth2 for Applications
- Weibo OAuth2
- Wunderlist OAuth2
- Xing OAuth1
- Yahoo OpenId and OAuth1
- Yammer OAuth2
- Yandex OAuth1, OAuth2 and OpenId

1.1.3 User data

Basic user data population, to allow custom fields values from providers response.

1.1.4 Social accounts association

Multiple social accounts can be associated to a single user.

1.1.5 Authentication and disconnection processing

Extensible pipeline to handle authentication, association and disconnection mechanism in ways that suits your project. Check Authentication Pipeline section.

CHAPTER 2

Installation

python-social-auth is very modular library looking to provide the basics tools to implement social authentication / authorization in Python projects. For that reason, the project is split in smaller components that focus on providing a simpler functionality. Some components are:

- social-auth-core Core library that the rest depends on, this contains the basic functionality to stablish an authentication/authorization flow with the diferent supported providers.
- social-auth-storage-sqlalchemy, social-auth-storage-peewee, social-auth-storage-mongoengine Different storage solutions that can be reused accross the supported frameworks or newer implementations.
- social-auth-app-django, social-auth-app-django-mongoengine Django framework integration
- social-auth-app-flask, social-auth-app-flask-sqlalchemy, social-auth-app-flask-mongoengine, social-auth-app-flask-peewee Flask framework integration
- social-auth-app-pyramid Pyramid framework integration
- social-auth-app-cherrypy Cherrypy framework integration
- social-auth-app-tornado Tornado framework integration
- social-auth-app-webpy Webpy framework integration

2.1 Dependencies

Dependencies are properly defined in the requirements files, the `setup.py` script will determine the environment where it's installed and sort between Python2 or Python3 packages if needed. There are some `extras` defined to install the corresponding dependencies since they require to build extensions that, unless used, are undesired.

- OpenIdConnect support requires the use of the `openidconnect` extra.
- SAML support requires the use of `saml` extra.

There's also the `all` extra that will install all the extra options.

Several backends demands application registration on their corresponding sites and other dependencies like sqlalchemy on Flask and Webpy.

2.2 Get a copy

From pypi:

```
$ pip install social-auth-<component>
```

Or:

```
$ easy_install social-auth-<component>
```

Or:

```
$ cd social-auth-<component>
$ sudo python setup.py install
```

2.3 Using the `extras` options

To enable any of the `extras` options to bring the dependencines for OpenIdConnect, or SAML, or both:

```
$ pip install "social-auth-core[openidconnect]"
$ pip install "social-auth-core[saml]"
$ pip install "social-auth-core[all]"
```

CHAPTER 3

Configuration

All the apps share the settings names, some settings for Django framework are special (like `AUTHENTICATION_BACKENDS`).

Below there's a main settings document detailing each configuration and its purpose, plus sections detailed for each framework and their particularities.

Support for more frameworks will be added in the future, pull-requests are very welcome.

Contents:

3.1 Configuration

3.1.1 Application setup

Once the application is installed (check Installation) define the following settings to enable the application behavior. Also check the sections dedicated to each framework for detailed instructions.

3.1.2 Settings name

Almost all settings are prefixed with `SOCIAL_AUTH_`, there are some exceptions for Django framework like `AUTHENTICATION_BACKENDS`.

All settings can be defined per-backend by adding the backend name to the setting name like `SOCIAL_AUTH_TWITTER_LOGIN_URL`. Settings discovery is done by reducing the name starting with backend setting, then app setting and finally global setting, for example:

```
SOCIAL_AUTH_TWITTER_LOGIN_URL
SOCIAL_AUTH_LOGIN_URL
LOGIN_URL
```

The backend name is generated from the `name` attribute from the backend class by uppercasing it and replacing `-` with `_`.

3.1.3 Keys and secrets

- Setup needed OAuth keys (see OAuth section for details):

```
SOCIAL_AUTH_TWITTER_KEY = 'foobar'
SOCIAL_AUTH_TWITTER_SECRET = 'bazqux'
```

OpenId backends don't require keys usually, but some need some API Key to call any API on the provider. Check Backends sections for details.

3.1.4 Authentication backends

Register the backends you plan to use, on Django framework use the usual `AUTHENTICATION_BACKENDS` settings, for others, define `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    'social_core.backends.open_id.OpenIdAuth',
    'social_core.backends.google.GoogleOpenId',
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.google.GoogleOAuth',
    'social_core.backends.twitter.TwitterOAuth',
    'social_core.backends.yahoo.YahooOpenId',
    ...
)
```

3.1.5 URLs options

These URLs are used on different steps of the auth process, some for successful results and others for error situations.

`SOCIAL_AUTH_LOGIN_REDIRECT_URL = '/logged-in/'` Used to redirect the user once the auth process ended successfully. The value of `?next=/foo` is used if it was present

`SOCIAL_AUTH_LOGIN_ERROR_URL = '/login-error/'` URL where the user will be redirected in case of an error

`SOCIAL_AUTH_LOGIN_URL = '/login-url/'` Is used as a fallback for `LOGIN_ERROR_URL`

`SOCIAL_AUTH_NEW_USER_REDIRECT_URL = '/new-users-redirect-url/'` Used to redirect new registered users, will be used in place of `SOCIAL_AUTH_LOGIN_REDIRECT_URL` if defined. Note that `?next=/foo` is appended if present, if you want new users to go to next, you'll need to do it yourself.

`SOCIAL_AUTH_NEW_ASSOCIATION_REDIRECT_URL = '/new-association-redirect-url/'` Like `SOCIAL_AUTH_NEW_USER_REDIRECT_URL` but for new associated accounts (user is already logged in). Used in place of `SOCIAL_AUTH_LOGIN_REDIRECT_URL`

`SOCIAL_AUTH_DISCONNECT_REDIRECT_URL = '/account-disconnected-redirect-url/'` The user will be redirected to this URL when a social account is disconnected

`SOCIAL_AUTH_INACTIVE_USER_URL = '/inactive-user/'` Inactive users can be redirected to this URL when trying to authenticate.

Successful URLs will default to `SOCIAL_AUTH_LOGIN_URL` while error URLs will fallback to `SOCIAL_AUTH_LOGIN_ERROR_URL`.

3.1.6 User model

`UserSocialAuth` instances keep a reference to the `User` model of your project, since this is not known, the `User` model must be configured by a setting:

```
SOCIAL_AUTH_USER_MODEL = 'foo.bar.User'
```

`User` model must have a `username` and `email` field, these are required.

Also an `is_authenticated` and `is_active` boolean flags are recommended, these can be methods if necessary (must return `True` or `False`). If the model lacks them a `True` value is assumed.

3.1.7 Tweaking some fields length

Some databases impose limitations on index columns (like MySQL InnoDB). These limitations won't play nice on some `UserSocialAuth` fields. To avoid such errors, define some of the following settings.

`SOCIAL_AUTH_UID_LENGTH = <int>` Used to define the max length of the field *uid*. A value of 223 should work when using MySQL InnoDB which impose a 767 bytes limit (assuming UTF-8 encoding).

`SOCIAL_AUTH_NONCE_SERVER_URL_LENGTH = <int>` `Nonce` model has a unique constraint over `('server_url', 'timestamp', 'salt')`, salt has a max length of 40, so `server_url` length must be tweaked using this setting.

`SOCIAL_AUTH_ASSOCIATION_SERVER_URL_LENGTH = <int>` or `SOCIAL_AUTH_ASSOCIATION_HANDLE_LENGTH =` `Association` model has a unique constraint over `('server_url', 'handle')`, both fields lengths can be tweaked by these settings.

3.1.8 Username generation

Some providers return a username, others just an ID or email or first and last names. The application tries to build a meaningful username when possible but defaults to generating one if needed.

A UUID is appended to usernames in case of collisions. Here are some settings to control username generation.

`SOCIAL_AUTH_UUID_LENGTH = 16` This controls the length of the UUID appended to usernames.

`SOCIAL_AUTH_USERNAME_IS_FULL_EMAIL = True` If you want to use the full email address as the `username`, define this setting.

`SOCIAL_AUTH_SLUGIFY_USERNAMES = False` For those that prefer slugged usernames, the `get_username` pipeline can apply a slug transformation (code borrowed from Django project) by defining this setting to `True`. The feature is disabled by default to to not force this option to all projects.

`SOCIAL_AUTH_CLEAN_USERNAMES = True` By default a set of regular expressions are applied over usernames to clean them from usual undesired characters like spaces. Set this setting to `False` to disable this behavior.

3.1.9 Extra arguments on auth processes

Some providers accept particular GET parameters that produce different results during the auth process, usually used to show different dialog types (mobile version, etc).

You can send extra parameters on auth process by defining settings per backend, example to request Facebook to show Mobile authorization page, define:

```
FACEBOOK_AUTH_EXTRA_ARGUMENTS = {'display': 'touch'}
```

For other providers, just define settings in the form:

```
SOCIAL_AUTH_<uppercase backend name>_AUTH_EXTRA_ARGUMENTS = {...}
```

Also, you can send extra parameters on request token process by defining settings in the same way explained above but with this other suffix:

```
SOCIAL_AUTH_<uppercase backend name>_REQUEST_TOKEN_EXTRA_ARGUMENTS = {...}
```

Basic information is requested to the different providers in order to create a coherent user instance (with first and last name, email and full name), this could be too intrusive for some sites that want to ask users the minimum data possible. It's possible to override the default values requested by defining any of the following settings, for Open Id providers:

```
SOCIAL_AUTH_<BACKEND_NAME>_IGNORE_DEFAULT_AX_ATTRS = True
SOCIAL_AUTH_<BACKEND_NAME>_AX_SCHEMA_ATTRS = [
    (schema, alias)
]
```

For OAuth backends:

```
SOCIAL_AUTH_<BACKEND_NAME>_IGNORE_DEFAULT_SCOPE = True
SOCIAL_AUTH_<BACKEND_NAME>_SCOPE = [
    ...
]
```

3.1.10 Processing redirects and urlopen

The application issues several redirects and API calls. The following settings allow some tweaks to the behavior of these.

SOCIAL_AUTH_SANITIZE_REDIRECTS = False The auth process finishes with a redirect, by default it's done to the value of `SOCIAL_AUTH_LOGIN_REDIRECT_URL` but can be overridden with `next` GET argument. If this setting is `True`, this application will vary the domain of the final URL and only redirect to it if it's on the same domain.

SOCIAL_AUTH_REDIRECT_IS_HTTPS = False On projects behind a reverse proxy that uses HTTPS, the redirect URIs can have the wrong schema (`http://` instead of `https://`) if the request lacks the appropriate headers, which might cause errors during the auth process. To force HTTPS in the final URIs set this setting to `True`

SOCIAL_AUTH_URLOPEN_TIMEOUT = 30 Any `urllib2.urlopen` call will be performed with the default timeout value, to change it without affecting the global socket timeout define this setting (the value specifies timeout seconds).

`urllib2.urlopen` uses `socket.getdefaulttimeout()` value by default, so setting `socket.setdefaulttimeout(...)` will affect `urlopen` when this setting is not defined, otherwise this setting takes precedence. Also this might affect other places in Django.

`timeout` argument was introduced in python 2.6 according to urllib2 documentation

3.1.11 Whitelists

Registration can be limited to a set of users identified by their email address or domain name. To white-list just set any of these settings:

SOCIAL_AUTH_<BACKEND_NAME>_WHITELISTED_DOMAINS = ['foo.com', 'bar.com'] Supply a list of domain names to be white-listed. Any user with an email address on any of the allowed domains will login successfully, otherwise `AuthForbidden` is raised.

SOCIAL_AUTH_<BACKEND_NAME>_WHITELISTED_EMAILS = ['me@foo.com', 'you@bar.com'] Supply a list of email addresses to be white-listed. Any user with an email address in this list will login successfully, otherwise `AuthForbidden` is raised.

3.1.12 Miscellaneous settings

SOCIAL_AUTH_PROTECTED_USER_FIELDS = ['email',] During the pipeline process a `dict` named `details` will be populated with the needed values to create the user instance, but it's also used to update the user instance. Any value in it will be checked as an attribute in the user instance (first by doing `hasattr(user, name)`). Usually there are attributes that cannot be updated (like `username`, `id`, `email`, etc.), those fields need to be *protect*. Set any field name that requires *protection* in this setting, and it won't be updated.

SOCIAL_AUTH_SESSION_EXPIRATION = False By default, user session expiration time will be set by your web framework (in Django, for example, it is set with SESSION_COOKIE_AGE). Some providers return the time that the access token will live, which is stored in `UserSocialAuth.extra_data` under the key `expires`. Changing this setting to True will override your web framework's session length setting and set user session lengths to match the `expires` value from the auth provider.

SOCIAL_AUTH_OPENID_PAPE_MAX_AUTH_AGE = <int value> Enable OpenID PAPE extension support by defining this setting.

SOCIAL_AUTH_FIELDS_STORED_IN_SESSION = ['foo',] If you want to store extra parameters from POST or GET in session, like it was made for `next` parameter, define this setting with the parameter names.

In this case `foo` field's value will be stored when user follows this link `<a href="{% url socialauth_begin 'github' %}?foo=bar">...</a>`.

SOCIAL_AUTH_PASSWORDLESS = False When this setting is `True` and `social_core.pipeline.mail.send_validation` is enabled, it allows the implementation of a passwordless authentication mechanism. Example of this implementation can be found at psa-passwordless.

SOCIAL_AUTH_USER_AGENT = None Define the *User-Agent* header value sent to on every request done to the service provider, used when combined with a backend that sets the *SEND_USER_AGENT* property to *True*. Default value is the string *social-auth-<version>*.

3.1.13 Account disconnection

Disconnect is an side-effect operation and should be done by POST method only, some CSRF protection is encouraged (and enforced on Django app). Ensure that any call to */disconnect/<backend>/* or */disconnect/<backend>/<id>/* is done using POST.

3.2 Django Framework

Django framework has a little more support since this application was derived from django-social-auth. Here are some details on configuring this application on Django.

3.2.1 Installing

From pypi:

```
$ pip install social-auth-app-django
```

And for MongoEngine ORM:

```
$ pip install social-auth-app-django-mongoengine
```

3.2.2 Register the application

The Django built-in app comes with two ORMs, one for default Django ORM and another for MongoEngine ORM.

Add the application to `INSTALLED_APPS` setting, for default ORM:

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

And for MongoEngine ORM:

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

Also ensure to define the MongoEngine storage setting:

```
SOCIAL_AUTH_STORAGE = 'social_django_mongoengine.models.DjangoStorage'
```

3.2.3 Database

(For Django 1.7 and higher) sync database to create needed models:

```
./manage.py migrate
```

3.2.4 Authentication backends

Add desired authentication backends to Django's AUTHENTICATION_BACKENDS setting:

```
AUTHENTICATION_BACKENDS = (
    'social_core.backends.open_id.OpenIdAuth',
    'social_core.backends.google.GoogleOpenId',
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.google.GoogleOAuth',
    'social_core.backends.twitter.TwitterOAuth',
    'social_core.backends.yahoo.YahooOpenId',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Take into account that backends **must** be defined in AUTHENTICATION_BACKENDS or Django won't pick them when trying to authenticate the user.

Don't miss `django.contrib.auth.backends.ModelBackend` if using `django.contrib.auth` application or users won't be able to login by username / password method.

3.2.5 URLs entries

Add URLs entries:

```
urlpatterns = patterns('',
    ...
    url('', include('social_django.urls', namespace='social'))
    ...
)
```

In case you need a custom namespace, this setting is also needed:

```
SOCIAL_AUTH_URL_NAMESPACE = 'social'
```

3.2.6 Templates

Example of google-oauth2 backend usage in template:

```
<a href="{% url "social:begin" "google-oauth2" %}">Google+</a>
```

3.2.7 Template Context Processors

There's a context processor that will add backends and associations data to template context:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social_django.context_processors.backends',
    'social_django.context_processors.login_redirect',
    ...
)
```

`backends` context processor will load a `backends` key in the context with three entries on it:

`associated` It's a list of `UserSocialAuth` instances related with the currently logged in user. Will be empty if there's no current user.

`not_associated` A list of available backend names not associated with the current user yet. If there's no user logged in, it will be a list of all available backends.

`backends` A list of all available backend names.

3.2.8 ORMs

As detailed above the built-in Django application supports default ORM and MongoEngine ORM.

When using MongoEngine make sure you've followed the instructions for MongoEngine Django integration, as you're now utilizing that user model. The *MongoEngine_* backend was developed and tested with version 0.6.10 of *MongoEngine_*.

Alternate storage models implementations currently follow a tight pattern of models that behave near or identical to Django ORM models. It is currently not decoupled from this pattern by any abstraction layer. If you would like to implement your own alternate, please see the `social_django.models` and `social_django_mongoengine.models` modules for guidance.

3.2.9 Exceptions Middleware

A base middleware is provided that handles `SocialAuthBaseException` by providing a message to the user via the Django messages framework, and then responding with a redirect to a URL defined in one of the middleware methods.

The middleware is at `social_django.middleware.SocialAuthExceptionMiddleware`. Any method can be overridden, but for simplicity these two are recommended:

```
get_message(request, exception)
get_redirect_uri(request, exception)
```

By default, the message is the exception message and the URL for the redirect is the location specified by the `LOGIN_ERROR_URL` setting.

If a valid backend was detected by `strategy()` decorator, it will be available at `request.strategy.backend` and `process_exception()` will use it to build a backend-dependent redirect URL but fallback to default if not defined.

Exception processing is disabled if any of this settings is defined with a `True` value:

```
<backend name>_SOCIAL_AUTH_RAISE_EXCEPTIONS = True
SOCIAL_AUTH_RAISE_EXCEPTIONS = True
RAISE_EXCEPTIONS = True
DEBUG = True
```

The redirect destination will get two `GET` parameters:

`message = ''` Message from the exception raised, in some cases it's the message returned by the provider during the auth process.

`backend = ''` Backend name that was used, if it was a valid backend.

The middleware will attempt to use the Django built-in *messages* application to store the exception message, and tag it with *social-auth* and the backend name. If the application is not enabled, or a *MessageFailure* error happens, the app will default to the URL format described above.

3.2.10 Django Admin

The default application (not the MongoEngine one) contains an `admin.py` module that will be auto-discovered by the usual mechanism.

But, by the nature of the application which depends on the existence of a user model, it's easy to fall in a recursive import ordering making the application fail to load. This happens because the admin module will build a set of fields to populate the `search_fields` property to search for related users in the administration UI, but this requires the user model to be retrieved which might not be defined at that time.

To avoid this issue define the following setting to circumvent the import error:

```
SOCIAL_AUTH_ADMIN_USER_SEARCH_FIELDS = ['field1', 'field2']
```

For example:

```
SOCIAL_AUTH_ADMIN_USER_SEARCH_FIELDS = ['username', 'first_name', 'email']
```

The fields listed **must** be user models fields.

It's also possible to define more search fields, not directly related to the user model by definig the following setting:

```
SOCIAL_AUTH_ADMIN_SEARCH_FIELDS = ['field1', 'field2']
```

3.3 Flask Framework

Flask reusable applications are tricky (or I'm not capable enough). Here are details on how to enable this application on Flask.

3.3.1 Dependencies

The *Flask app* does not depend on any storage backend by default. There's support for SQLAlchemy, MongoEngine and Peewee.

3.3.2 Installing

Install the flask core from pypi:

```
$ pip install social-auth-app-flask
```

Install any of the storage solutions:

```
$ pip install social-auth-app-flask-sqlalchemy
$ pip install social-auth-app-flask-mongoengine
$ pip install social-auth-app-flask-peewee
```

3.3.3 Enabling the application

The applications define a Flask Blueprint, which needs to be registered once the Flask app is configured by:

```
from social_flask.routes import social_auth

app.register_blueprint(social_auth)
```

For MongoEngine you need this setting:

```
SOCIAL_AUTH_STORAGE = 'social_flask_mongoengine.models.FlaskStorage'
```

For Peewee you need this setting:

```
SOCIAL_AUTH_STORAGE = 'social_flask_peewee.models.FlaskStorage'
```

3.3.4 Models Setup

At the moment the models for python-social-auth are defined inside a function because they need the reference to the current db session and the User model used on your project (check *User model reference* below). Once the Flask app and the database are defined, call `init_social` to register the models:

```
from social_flask_sqlalchemy.models import init_social

init_social(app, session)
```

For MongoEngine:

```
from social_flask_mongoengine.models import init_social

init_social(app, session)
```

For Peewee:

```
from social_flask_peewee.models import init_social

init_social(app, session)
```

So far I wasn't able to find another way to define the models on another way rather than making it as a side-effect of calling this function since the database is not available and `current_app` cannot be used on init time, just run time.

3.3.5 User model reference

The application keeps a reference to the User model used by your project, define it by using this setting:

```
SOCIAL_AUTH_USER_MODEL = 'foobar.models.User'
```

The value must be the import path to the User model.

3.3.6 Global user

The application expects the current logged in user accesible at `g.user`, define a handler like this to ensure that:

```
@app.before_request
def global_user():
    g.user = get_current_logged_in_user
```

3.3.7 Flask-Login

The application works quite well with Flask-Login, ensure to have some similar handlers to these:

```
@login_manager.user_loader
def load_user(userid):
    try:
        return User.query.get(int(userid))
    except (TypeError, ValueError):
        pass


@app.before_request
```

```
def global_user():
    g.user = login.current_user


# Make current user available on templates
@app.context_processor
def inject_user():
    try:
        return {'user': g.user}
    except AttributeError:
        return {'user': None}
```

3.3.8 Remembering sessions

The users session can be remembered when specified on login. The common implementation for this feature is to pass a parameter from the login form (`remember_me`, `keep`, etc), to flag the action. Flask-Login will mark the session as persistent if told so.

python-social-auth will check for a given name (`keep`) by default, but since providers won't pass parameters back to the application, the value must be persisted in the session before the authentication process happens.

So, the following setting is required for this to work:

```
SOCIAL_AUTH_FIELDS_STORED_IN_SESSION = ['keep']
```

It's possible to override the default name with this setting:

```
SOCIAL_AUTH_REMEMBER_SESSION_NAME = 'remember_me'
```

Don't use the value `remember` since that will clash with Flask-Login which pops the value from the session.

Then just pass the parameter `keep=1` as a GET or POST parameter.

3.3.9 Exceptions handling

The Django application has a middleware (that fits in the framework architecture) to facilitate the different exceptions handling raised by python-social-auth. The same can be accomplished (even on a simpler way) in Flask by defining an errorhandler. For example the next code will redirect any social-auth exception to a `/socialerror` URL:

```
from social_core.exceptions import SocialAuthBaseException


@app.errorhandler(500)
def error_handler(error):
    if isinstance(error, SocialAuthBaseException):
        return redirect('/socialerror')
```

Be sure to set your debug and test flags to `False` when testing this on your development environment, otherwise the exception will be raised and error handlers won't be called.

3.4 Pyramid Framework

Pyramid reusable applications are tricky (or I'm not capable enough). Here are details on how to enable this application on Pyramid.

3.4.1 Dependencies

The Pyramid app depends on sqlalchemy, there's no support for others ORMs yet but pull-requests are welcome.

3.4.2 Installing

From pypi:

```
$ pip install social-auth-app-pyramid
```

3.4.3 Enabling the application

The application can be scanned by `Configurator.scan()`, also it defines an `includeme()` in the `__init__.py` file which will add the needed routes to your application configuration. To scan it just add:

```
config.include('social_pyramid')
config.scan('social_pyramid')
```

3.4.4 Models Setup

At the moment the models for python-social-auth are defined inside a function because they need the reference to the current DB instance and the User model used on your project (check *User model reference* below). Once the Pyramid application configuration and database are defined, call `init_social` to register the models:

```
from social_pyramid.models import init_social

init_social(config, Base, DBSession)
```

So far I wasn't able to find another way to define the models on another way rather than making it as a side-effect of calling this function since the database is not available and `current_app` cannot be used on initialization time, just run time.

3.4.5 User model reference

The application keeps a reference to the User model used by your project, define it by using this setting:

```
SOCIAL_AUTH_USER_MODEL = 'foobar.models.User'
```

The value must be the import path to the User model.

3.4.6 Global user

The application expects the current logged in user accessible at `request.user`, the example application ensures that with this hander:

```
def get_user(request):
    user_id = request.session.get('user_id')
    if user_id:
        user = DBSession.query(User)\
                        .filter(User.id == user_id)\
```

```
                        .first()
    else:
        user = None
    return user
```

The handler is added to the configuration doing:

```
config.add_request_method('example.auth.get_user', 'user', reify=True)
```

This is just a simple example, probably your project does it in a better way.

3.4.7 User login

Since the application doesn't make any assumption on how you are going to login the users, you need to specify it. In order to do that, define these settings:

```
SOCIAL_AUTH_LOGIN_FUNCTION = 'example.auth.login_user'
SOCIAL_AUTH_LOGGEDIN_FUNCTION = 'example.auth.login_required'
```

The first one must accept the strategy used and the user instance that was created or retrieved from the database, there you can set the user id in the session or cookies or whatever place used later to retrieve the id again and load the user from the database (check the snippet above in *Global User*).

The second one is used to ensure that there's a user logged in when calling the disconnect view. It must accept a `User` instance and return `True` or `Flase`.

Check the auth.py in the example application for details on how it's done there.

3.4.8 Social auth in templates context

To access the social instances related to a user in the template context, you can do so by accessing the `social_auth` attribute in the user instance:

```
<li tal:repeat="social request.user.social_auth">${social.provider}</li>
```

Also you can add the backends (associated and not associated to a user) by enabling this context function in your project:

```
from pyramid.events import subscriber, BeforeRender
from social_pyramid.utils import backends

@subscriber(BeforeRender)
def add_social(event):
    request = event['request']
    event.update(backends(request, request.user))
```

That will load a dict with entries:

```
{
    'associated': [...],
    'not_associated': [...],
    'backends': [...]
}
```

The `associated` key will have all the associated `UserSocialAuth` instances related to the given user. `not_associated` will have the backends names not associated and backends will have all the enabled backends names.

3.5 CherryPy Framework

CherryPy framework is supported, it works but I'm sure there's room for improvements. The implementation uses SQLAlchemy as ORM and expects some values accessible on `cherrypy.request` for it to work.

At the moment the configuration is expected on `cherrypy.config` but ideally it should be an application configuration instead.

Expected values are:

`cherrypy.request.user` Current logged in user, load it in your application on a `before_handler` handler.

`cherrypy.request.db` Current database session, again, load it in your application on a `before_handler`.

3.5.1 Dependencies

The *CherryPy application* depends on sqlalchemy, there's no support for others ORMs yet.

3.5.2 Installing

From pypi:

```
$ pip install social-auth-app-cherrypy
```

3.5.3 Enabling the application

The application is defined on `social_cherrypy.views.CherryPyPSAViews`, register it in the preferred way for your project.

Check the rest of the docs for the other settings like enabling authentication backends and backends keys.

3.5.4 Models Setup

The models are located in `social_cherrypy.models`. A reference to your `User` model is required to be defined in the project settings, it should be an import path, for example:

```
cherrypy.config.update({
    'SOCIAL_AUTH_USER_MODEL': 'models.User'
})
```

3.5.5 Login mechanism

By default the application sets the session value `user_id`, this is a simple solution and it should be improved, if you want to provider your own login mechanism you can do it by defining the `SOCIAL_AUTH_LOGIN_METHOD` setting, it should be an import path to a callable, like this:

```
SOCIAL_AUTH_USER_MODEL = 'app.login_user'
```

And an example of this function:

```
def login_user(strategy, user):
    strategy.session_set('user_id', user.id)
```

Then, ensure to load the user in your application at `cherrypy.request.user`, for example:

```
def load_user():
    user_id = cherrypy.session.get('user_id')
    if user_id:
        cherrypy.request.user = cherrypy.request.db.query(User).get(user_id)
    else:
        cherrypy.request.user = None


cherrypy.tools.authenticate = cherrypy.Tool('before_handler', load_user)
```

3.6 Webpy Framework

Webpy framework is easy to setup, once that python-social-auth is installed or accessible in the `PYTHONPATH`, just add the needed configurations to make it run.

3.6.1 Dependencies

The *Webpy app* depends on sqlalchemy, there's no support for others ORMs yet but pull-requests are welcome.

3.6.2 Installing

From pypi:

```
$ pip install social-auth-app-webpy
```

3.6.3 Configuration

Add the needed settings into `web.config` store. Settings are prefixed with `SOCIAL_AUTH_` but there's a helper for it:

```
from social_core.utils import setting_name

web.config[setting_name('USER_MODEL')] = 'models.User'
web.config[setting_name('LOGIN_REDIRECT_URL')] = '/done/'
web.config[setting_name('AUTHENTICATION_BACKENDS')] = (
    'social_core.backends.google.GoogleOAuth2',
    ...
)
```

Add all the settings needed for the app (check *Configuration* section for details).

3.6.4 URLs

Add the social application into URLs:

```
from social_webpy import app as social_app

urls = (
    ...
    '', social_app.app_social
    ...
)
```

3.6.5 Session

python-social-auth depends on sessions storage to keep some essential values, usually redirects and `state` parameters used to validate authentication process on OAuth providers.

The *Webpy built-in app* expects the session reference to be available under `web.web_session` so ensure it's available there.

3.6.6 User model

Like the other apps, the User model must be defined on settings since a reference to it is kept on `UserSocialAuth` instance. Define like this:

```
web.config[setting_name('USER_MODEL')] = 'models.User'
```

Where the value is the import path to the User model used on your project.

3.7 Porting from django-social-auth

Being a derivative work from django-social-auth, porting from it to python-social-auth should be an easy task. Porting to others libraries usually is a pain, I'm trying to make this as easy as possible.

3.7.1 Installed apps

On django-social-auth there was a single application to add into `INSTALLED_APPS` plus a setting to define which ORM to be used (default or MongoEngine). Now the apps are split and there's not need for that extra setting.

When using the default ORM:

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

And when using MongoEngine:

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

The models table names were defined to be compatible with those used on django-social-auth, so data is not needed to be migrated.

3.7.2 URLs

The URLs are namespaced, you can chose your namespace, the example app uses the `social` namespace. Replace the old include with:

```
urlpatterns = patterns('',
    ...
    url('', include('social_django.urls', namespace='social'))
    ...
)
```

On templates use a namespaced URL:

```
{% url 'social:begin' "google-oauth2" %}
```

Account disconnection URL would be:

```
{% url 'social:disconnect_individual' provider, id %}
```

3.7.3 Porting settings

All python-social-auth settings are prefixed with `SOCIAL_AUTH_`, except for some exception on Django framework, `AUTHENTICATION_BACKENDS` remains the same for obvious reasons.

All backends settings have the backend name into it, all uppercase and with dashes replaced with underscores, take for instance Google OAuth2 backend is named `google-oauth2`, any setting name related to that backend should start with `SOCIAL_AUTH_GOOGLE_OAUTH2_`.

Keys and secrets are some mandatory settings needed for OAuth providers, to keep consistency the names follow the same naming convention `*_KEY` for the application key, and `*_SECRET` for the secret. OAuth1 backends use to have `CONSUMER` in the setting name, not anymore. Following with the Google OAuth2 example:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '...'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '...'
```

Remember that the name of the backend is needed in the settings, and names differ a little from backend to backend, like Facebook OAuth2 backend name is `facebook`. So the settings should be:

```
SOCIAL_AUTH_FACEBOOK_KEY = '...'
SOCIAL_AUTH_FACEBOOK_SECRET = '...'
```

3.7.4 Authentication backends

Import path for authentication backends changed a little, there's no more `contrib` module, there's no need for it. Some backends changed the names to have some consistency, check the backends, it should be easy to track the names

changes. Examples of the new import paths:

```
AUTHENTICATION_BACKENDS = (
    'social_core.backends.open_id.OpenIdAuth',
    'social_core.backends.google.GoogleOpenId',
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.google.GoogleOAuth',
    'social_core.backends.twitter.TwitterOAuth',
    'social_core.backends.facebook.FacebookOAuth2',
)
```

3.7.5 Session

Django stores the last authentication backend used in the user session as an import path, this can cause import troubles when porting since the old import paths aren't valid anymore. Some solutions to this problem are:

1. Clean the session and force the users to login again in your site
2. Run a migration script that will update the authentication backend session value for each session in your database. This implies figuring out the new import path for each backend you have configured, which is the value used in `AUTHENTICATION_BACKENDS` setting.

   @tomgruner created a Gist here that updates the value just for Facebook backend. A `template` for this script would look like this:

```
from django.contrib.sessions.models import Session

BACKENDS = {
    'social_auth.backends.facebook.FacebookBackend': 'social_core.backends.
↪facebook.FacebookOAuth2'
}

for sess in Session.objects.iterator():
    session_dict = sess.get_decoded()

    if '_auth_user_backend' in session_dict.keys():
        # Change old backend import path from new backend import path
        if session_dict['_auth_user_backend'].startswith('social_auth'):
            session_dict['_auth_user_backend'] = BACKENDS[session_dict['_auth_
↪user_backend']]
            new_sess = Session.objects.save(sess.session_key, session_dict, sess.
↪expire_date)
            print 'New session saved {}'.format(new_sess.pk)
```

CHAPTER 4

Pipeline

python-social-auth uses an extendible pipeline mechanism where developers can introduce their functions during the authentication, association and disconnection flows.

The functions will receive a variable set of arguments related to the current process, common arguments are the current `strategy`, `user` (if any) and `request`. It's recommended that all the function also define an `**kwargs` in the parameters to avoid errors for unexpected arguments.

Each pipeline entry can return a `dict` or `None`, any other type of return value is treated as a response instance and returned directly to the client, check *Partial Pipeline* below for details.

If a `dict` is returned, the value in the set will be merged into the `kwargs` argument for the next pipeline entry, `None` is taken as if `{}` was returned.

4.1 Authentication Pipeline

The final process of the authentication workflow is handled by an operations pipeline where custom functions can be added or default items can be removed to provide a custom behavior. The default pipeline is a mechanism that creates user instances and gathers basic data from providers.

The default pipeline is composed by:

```
(
    # Get the information we can about the user and return it in a simple
    # format to create the user instance later. On some cases the details are
    # already part of the auth response from the provider, but sometimes this
    # could hit a provider API.
    'social_core.pipeline.social_auth.social_details',

    # Get the social uid from whichever service we're authing thru. The uid is
    # the unique identifier of the given user in the provider.
    'social_core.pipeline.social_auth.social_uid',

    # Verifies that the current auth process is valid within the current
    # project, this is where emails and domains whitelists are applied (if
```

```
    # defined).
    'social_core.pipeline.social_auth.auth_allowed',

    # Checks if the current social-account is already associated in the site.
    'social_core.pipeline.social_auth.social_user',

    # Make up a username for this person, appends a random string at the end if
    # there's any collision.
    'social_core.pipeline.user.get_username',

    # Send a validation email to the user to verify its email address.
    # Disabled by default.
    # 'social_core.pipeline.mail.mail_validation',

    # Associates the current social details with another user account with
    # a similar email address. Disabled by default.
    # 'social_core.pipeline.social_auth.associate_by_email',

    # Create a user account if we haven't found one yet.
    'social_core.pipeline.user.create_user',

    # Create the record that associates the social account with the user.
    'social_core.pipeline.social_auth.associate_user',

    # Populate the extra_data field in the social record with the values
    # specified by settings (and the default ones like access_token, etc).
    'social_core.pipeline.social_auth.load_extra_data',

    # Update the user record with any changed info from the auth service.
    'social_core.pipeline.user.user_details',
)
```

It's possible to override it by defining the setting `SOCIAL_AUTH_PIPELINE`. For example, a pipeline that won't create users, just accept already registered ones would look like this:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

Note that this assumes the user is already authenticated, and thus the `user` key in the dict is populated. In cases where the authentication is purely external, a pipeline method must be provided that populates the `user` key. Example:

```
SOCIAL_AUTH_PIPELINE = (
    'myapp.pipeline.load_user',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

It is also possible to define pipelines on a per backend basis by defining a setting such as `SOCIAL_AUTH_TWITTER_PIPELINE`. Backend specific pipelines will override the non specific pipelines

(i.e. the default pipeline and `SOCIAL_AUTH_PIPELINE`).

Each pipeline function will receive the following parameters:

- Current strategy (which gives access to current store, backend and request)
- User ID given by authentication provider
- User details given by authentication provider
- `is_new` flag (initialized as `False`)
- Any arguments passed to `auth_complete` backend method, default views pass these arguments:
  - current logged in user (if it's logged in, otherwise `None`)
  - current request

4.2 Disconnection Pipeline

Like the authentication pipeline, it's possible to define a disconnection pipeline if needed.

For example, this can be useful on sites where a user that disconnects all the related social account is required to fill a password to ensure the authentication process in the future. This can be accomplished by overriding the default disconnection pipeline and setup a function that checks if the user has a password, in case it doesn't a redirect to a fill-your-password form can be returned and later continue the disconnection process, take into account that disconnection ensures the POST method by default, a simple method to ensure this, is to make your form POST to `/disconnect/` and set the needed password in your pipeline function. Check *Partial Pipeline* below.

In order to override the disconnection pipeline, just define the setting:

```
SOCIAL_AUTH_DISCONNECT_PIPELINE = (
    # Verifies that the social association can be disconnected from the current
    # user (ensure that the user login mechanism is not compromised by this
    # disconnection).
    'social_core.pipeline.disconnect.allowed_to_disconnect',

    # Collects the social associations to disconnect.
    'social_core.pipeline.disconnect.get_entries',

    # Revoke any access_token when possible.
    'social_core.pipeline.disconnect.revoke_tokens',

    # Removes the social associations.
    'social_core.pipeline.disconnect.disconnect',
)
```

Backend specific disconnection pipelines can also be defined with a setting such as `SOCIAL_AUTH_TIWTTER_DISCONNECT_PIPELINE`.

4.3 Partial Pipeline

It's possible to cut the pipeline process to return to the user asking for more data and resume the process later. To accomplish this decorate the function that will cut the process with the `@partial` decorator located at `social/pipeline/partial.py`.

The old `social_core.pipeline.partial.save_status_to_session` is now deprecated.

When it's time to resume the process just redirect the user to `/complete/<backend>/` or `/disconnect/<backend>/` view. The pipeline will resume in the same function that cut the process.

`@partial` and `save_status_to_session` stores needed data into user session under the key `partial_pipeline`. To get the backend in order to redirect to any social view, just do:

```
backend = session['partial_pipeline']['backend']
```

Check the example applications to check a basic usage.

4.4 Email validation

There's a pipeline to validate email addresses, but it relies a lot on your project.

The pipeline is at `social_core.pipeline.mail.mail_validation` and it's a partial pipeline, it will return a redirect to a URL that you can use to tell the users that an email validation was sent to them. If you want to mention the email address you can get it from the session under the key `email_validation_address`.

In order to send the validation python-social-auth needs a function that will take care of it, this function is defined by the developer with the setting `SOCIAL_AUTH_EMAIL_VALIDATION_FUNCTION`. It should be an import path. This function should take three arguments `strategy`, `backend` and `code`. `code` is a model instance used to validate the email address, it contains three fields:

`code = '...'` Holds an `uuid.uuid4()` value and it's the code used to identify the validation process.

`email = '...'` Email address trying to be validate.

`verified = True / False` Flag marking if the email was verified or not.

You should use the code in this instance to build the link for email validation which should go to `/complete/email?verification_code=<code here>`. If you are using Django, you can do it with:

```
from django.core.urlresolvers import reverse
url = strategy.build_absolute_uri(
    reverse('social:complete', args=(strategy.backend_name,))
) + '?verification_code=' + code.code
```

On Flask:

```
from flask import url_for
url = url_for('social.complete', backend=strategy.backend_name,
              _external=True) + '?verification_code=' + code
```

This pipeline can be used globally with any backend if this setting is defined:

```
SOCIAL_AUTH_FORCE_EMAIL_VALIDATION = True
```

Or individually by defining the setting per backend basis like `SOCIAL_AUTH_TWITTER_FORCE_EMAIL_VALIDATION = True`.

CHAPTER 5

Extending the Pipeline

The main purpose of the pipeline (either creation or deletion pipelines) is to allow extensibility for developers. You can jump in the middle of it, do changes to the data, create other models instances, ask users for extra data, or even halt the whole process.

Extending the pipeline implies:

1. Writing a function
2. Locating the function in an accessible path (accessible in the way that it can be imported)
3. Overriding the default pipeline definition with one that includes newly created function.

The part of writing the function is quite simple. However please be careful when placing your function in the pipeline definition, because order does matter in this case! Ordering of functions in `SOCIAL_AUTH_PIPELINE` will determine the value of arguments that each function will receive. For example, adding your function after `social_core.pipeline.user.create_user` ensures that your function will get the user instance (created or already existent) instead of a `None` value.

The pipeline functions will get quite a lot of arguments, ranging from the backend in use, different model instances, server requests and provider responses. To enumerate a few:

`strategy` The current strategy instance.

`backend` The current backend instance.

`uid` User ID in the provider, this `uid` should identify the user in the current provider.

`response = {} or object()` The server user-details response, it depends on the protocol in use (and sometimes the provider implementation of such protocol), but usually it's just a `dict` with the user profile details in such provider. Lots of information related to the user is provided here, sometimes the `scope` will increase the amount of information in this response on OAuth providers.

`details = {}` Basic user details generated by the backend, used to create/update the user model details (this `dict` will contain values like `username`, `email`, `first_name`, `last_name` and `fullname`).

`user = None` The user instance (or `None` if it wasn't created or retrieved from the database yet).

`social = None` This is the associated `UserSocialAuth` instance for the given user (or `None` if it wasn't created or retrieved from the DB yet).

Usually when writing your custom pipeline function, you just want to get some values from the `response` parameter. But you can do even more, like call other APIs endpoints to retrieve even more details about the user, store them on some other place, etc.

Here's an example of a simple pipeline function that will create a `Profile` class instance, related to the current user. This profile will store some simple details returned by the provider (`Facebook` in this example). The usual Facebook `response` looks like this:

```
{
    'username': 'foobar',
    'access_token': 'CAAD...',
    'first_name': 'Foo',
    'last_name': 'Bar',
    'verified': True,
    'name': 'Foo Bar',
    'locale': 'en_US',
    'gender': 'male',
    'expires': '5183999',
    'email': 'foo@bar.com',
    'updated_time': '2014-01-14T15:58:35+0000',
    'link': 'https://www.facebook.com/foobar',
    'timezone': -3,
    'id': '100000126636010',
}
```

Let's say we are interested in storing the user profile link, the gender and the timezone in our `Profile` model:

```
def save_profile(backend, user, response, *args, **kwargs):
    if backend.name == 'facebook':
        profile = user.get_profile()
        if profile is None:
            profile = Profile(user_id=user.id)
        profile.gender = response.get('gender')
        profile.link = response.get('link')
        profile.timezone = response.get('timezone')
        profile.save()
```

Now all that's needed is to tell `python-social-auth` to use our function in the pipeline. Since the function uses user instance, we need to put it after `social_core.pipeline.user.create_user`:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'path.to.save_profile',  # <--- set the path to the function
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

So far the function we created returns `None`, which is taken as if `{}` was returned. If you want the `profile` object to be available to the next function in the pipeline, all you need to do is return `{'profile': profile}`.

CHAPTER 6

Strategies

Different strategies are defined to encapsulate the different frameworks capabilities under a common API to reuse as much code as possible.

6.1 Description

A strategy's responsibility is to provide access to:

- Request data and host information and URI building
- Session access
- Project settings
- Response types (HTML and redirects)
- HTML rendering

Different frameworks implement these features on different ways, thus the need for these interfaces.

6.2 Implementing a new Strategy

The following methods must be defined on strategies sub-classes.

Request:

```
def request_data(self):
    """Return current request data (POST or GET)"""
    raise NotImplementedError('Implement in subclass')

def request_host(self):
    """Return current host value"""
    raise NotImplementedError('Implement in subclass')
```

```
def build_absolute_uri(self, path=None):
    """Build absolute URI with given (optional) path"""
    raise NotImplementedError('Implement in subclass')
```

Session:

```
def session_get(self, name):
    """Return session value for given key"""
    raise NotImplementedError('Implement in subclass')

def session_set(self, name, value):
    """Set session value for given key"""
    raise NotImplementedError('Implement in subclass')

def session_pop(self, name):
    """Pop session value for given key"""
    raise NotImplementedError('Implement in subclass')
```

Settings:

```
def get_setting(self, name):
    """Return value for given setting name"""
    raise NotImplementedError('Implement in subclass')
```

Responses:

```
def html(self, content):
    """Return HTTP response with given content"""
    raise NotImplementedError('Implement in subclass')

def redirect(self, url):
    """Return a response redirect to the given URL"""
    raise NotImplementedError('Implement in subclass')

def render_html(self, tpl=None, html=None, context=None):
    """Render given template or raw html with given context"""
    raise NotImplementedError('Implement in subclass')
```

CHAPTER 7

Storage

Different frameworks support different ORMs, Storage solves the different interfaces moving the common API to mixins classes. These mixins are used on apps when defining the different models used by `python-social-auth`.

7.1 Social User

This model associates a social account data with a user in the system, it contains the provider name and user ID (`uid`) which should identify the social account in the remote provider, plus some extra data (`extra_data`) which is JSON encoded field with extra information from the provider (usually avatars and similar).

When implementing this model, it must inherits from UserMixin and extend the needed methods:

- Username:

```
@classmethod
def get_username(cls, user):
    """Return the username for given user"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def username_max_length(cls):
    """Return the max length for username"""
    raise NotImplementedError('Implement in subclass')
```

- User model:

```
@classmethod
def user_model(cls):
    """Return the user model"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def changed(cls, user):
    """The given user instance is ready to be saved"""
```

```
    raise NotImplementedError('Implement in subclass')

@classmethod
def user_exists(cls, username):
    """
    Return True/False if a User instance exists with the given arguments.
    Arguments are directly passed to filter() manager method.
    """
    raise NotImplementedError('Implement in subclass')

@classmethod
def create_user(cls, username, email=None):
    """Create a user with given username and (optional) email"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def get_user(cls, pk):
    """Return user instance for given id"""
    raise NotImplementedError('Implement in subclass')
```

- Social user:

```
@classmethod
def get_social_auth(cls, provider, uid):
    """Return UserSocialAuth for given provider and uid"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def get_social_auth_for_user(cls, user):
    """Return all the UserSocialAuth instances for given user"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def create_social_auth(cls, user, uid, provider):
    """Create a UserSocialAuth instance for given user"""
    raise NotImplementedError('Implement in subclass')
```

- Social disconnection:

```
@classmethod
def allowed_to_disconnect(cls, user, backend_name, association_id=None):
    """Return if it's safe to disconnect the social account for the
    given user"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def disconnect(cls, name, user, association_id=None):
    """Disconnect the social account for the given user"""
    raise NotImplementedError('Implement in subclass')
```

7.2 Nonce

This is a helper class for OpenId mechanism, it stores a one-use number, shouldn't be used by the project since it's for internal use only.

When implementing this model, it must inherits from NonceMixin, and override the needed method:

```
@classmethod
def use(cls, server_url, timestamp, salt):
    """Create a Nonce instance"""
    raise NotImplementedError('Implement in subclass')
```

7.3 Association

Another OpenId helper class, it stores basic data to keep the OpenId association. Like *Nonce* this is for internal use only.

When implementing this model, it must inherits from AssociationMixin, and override the needed methods:

```
@classmethod
def store(cls, server_url, association):
    """Create an Association instance"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def get(cls, *args, **kwargs):
    """Get an Association instance"""
    raise NotImplementedError('Implement in subclass')

@classmethod
def remove(cls, ids_to_delete):
    """Remove an Association instance"""
    raise NotImplementedError('Implement in subclass')
```

7.4 Validation code

This class is used to keep track of email validations codes following the usual email validation mechanism of sending an email to the user with a unique code. This model is used by the partial pipeline `social_core.pipeline.mail.mail_validation`. Check the docs at *Email validation* in pipeline docs.

When implementing the model for your framework only one method needs to be overridden:

```
@classmethod
def get_code(cls, code):
    """Return the Code instance with the given code value"""
    raise NotImplementedError('Implement in subclass')
```

7.5 Storage interface

There's a helper class used by strategies to hide the real models names under a common API, an instance of this class is used by strategies to access the storage modules.

When implementing this class it must inherits from BaseStorage, add the needed models references and implement the needed method:

```
class StorageImplementation(BaseStorage):
    user = UserModel
    nonce = NonceModel
```

```
    association = AssociationModel
    code = CodeModel

    @classmethod
    def is_integrity_error(cls, exception):
        """Check if given exception flags an integrity error in the DB"""
        raise NotImplementedError('Implement in subclass')
```

7.6 SQLAlchemy and Django mixins

Currently there are partial implementations of mixins for SQLAlchemy ORM and Django ORM with common code used later on current implemented applications.

Note: When using SQLAlchemy ORM and `ZopeTransactionExtension`, it's recommended to use the transaction application to handle them.

7.7 Models Examples

Check for current implementations for Django App, Flask App, Pyramid App, and Webpy App for examples of implementations.

CHAPTER 8

Exceptions

This set of exceptions were introduced to describe the situations a bit more than just the `ValueError` usually raised.

`SocialAuthBaseException` Base class for all social auth exceptions.

`AuthException` Base exception class for authentication process errors.

`AuthFailed` Authentication failed for some reason.

`AuthCanceled` Authentication was canceled by the user.

`AuthUnknownError` An unknown error stoped the authentication process.

`AuthTokenError` Unauthorized or access token error, it was invalid, impossible to authenticate or user removed permissions to it.

`AuthMissingParameter` A needed parameter to continue the process was missing, usually raised by the services that need some POST data like myOpenID.

`AuthAlreadyAssociated` A different user has already associated the social account that the current user is trying to associate.

`WrongBackend` Raised when the backend given in the URLs is invalid (not enabled or registered).

`NotAllowedToDisconnect` Raised on disconnect action when it's not safe for the user to disconnect the social account, probably because the user lacks a password or another social account.

`AuthStateMissing` The state parameter is missing from the server response.

`AuthStateForbidden` The state parameter returned by the server is not the one sent.

`AuthTokenRevoked` Raised when the user revoked the access_token in the provider.

`AuthUnreachableProvider` Raised when server couldn't communicate with backend.

These are a subclass of `ValueError` to keep backward compatibility.

CHAPTER 9

Backends

Here's a list and detailed instruction on how to setup the support for each backend.

9.1 Adding new backend support

Add new backends is quite easy, usually adding just a `class` with a couple methods overrides to retrieve user data from services API. Follow the details in the *Implementation* docs.

9.1.1 Adding a new backend

Add new backends is quite easy, usually adding just a `class` with a couple settings and methods overrides to retrieve user data from services API. Follow the details below.

Common attributes

First, lets check the common attributes for all backend types.

`name = ''` Any backend needs a name, usually the popular name of the service is used, like `facebook`, `twitter`, etc. It must be unique, otherwise another backend can take precedence if it's listed before in `AUTHENTICATION_BACKENDS` setting.

`ID_KEY = None` Defines the attribute in the service response that identifies the user as unique in the service, the value is later stored in the `uid` attribute in the `UserSocialAuth` instance.

`REQUIRES_EMAIL_VALIDATION = False` Flags the backend to enforce email validation during the pipeline (if the corresponding pipeline `social_core.pipeline.mail.mail_validation` was enabled).

`EXTRA_DATA = None` During the auth process some basic user data is returned by the provider or retrieved by `user_data()` method which usually is used to call some API on the provider to retrieve it. This data will be stored under `UserSocialAuth.extra_data` attribute, but to make it accessible under some common names on different providers, this attribute defines a list of tuples in the form `(name, alias)` where `name` is the key in the user data (which should be a `dict` instance) and `alias` is the name to store it on `extra_data`.

OAuth

OAuth1 and OAuth2 provide share some common definitions based on the shared behavior during the auth process, like a successful API response from `AUTHORIZATION_URL` usually returns some basic user data like a user Id.

Shared attributes

`name` This defines the backend name and identifies it during the auth process. The name is used in the URLs `/login/<backend name>` and `/complete/<backend name>`.

`ID_KEY = 'id'` Default key name where user identification field is defined, it's used on auth process when some basic user data is returned. This Id is stored in `UserSocialAuth.uid` field, this together the `UserSocialAuth.provider` field is used to unique identify a user association.

`SCOPE_PARAMETER_NAME = 'scope'` Scope argument is used to tell the provider the API endpoints you want to call later, it's a permissions request granted over the `access_token` later retrieved. Default value is `scope` since that's usually the name used in the URL parameter, but can be overridden if needed.

`DEFAULT_SCOPE = None` Some providers give nothing about the user but some basic data in required like the user Id or an email address. Default scope attribute is used to specify a default value for `scope` argument to request those extra used bits.

`SCOPE_SEPARATOR = ' '` The `scope` argument is usually a list of permissions to request, the list is joined used a separator, usually just a blank space, but differ from provider to provider, override the default value with this attribute if it differs.

OAuth2

OAuth2 backends are fairly simple to implement; just a few settings, a method override and it's mostly ready to go.

The key points on this backends are:

`AUTHORIZATION_URL` This is the entry point for the authorization mechanism, users must be redirected to this URL, used on `auth_url` method which builds the redirect address with `AUTHORIZATION_URL` plus some arguments (`client_id`, `redirect_uri`, `response_type`, and `state`).

`ACCESS_TOKEN_URL` Must point to the API endpoint that provides an `access_token` needed to authenticate in users behalf on future API calls.

`REFRESH_TOKEN_URL` Some providers give the option to renew the `access_token` since they are usually limited in time, once that time runs out, the token is invalidated and cannot be used any more. This attribute should point to that API endpoint.

`RESPONSE_TYPE` The response type expected on the auth process, default value is `code` as dictated by OAuth2 definition. Override it if default value doesn't fit the provider implementation.

`STATE_PARAMETER` OAuth2 defines that an `state` parameter can be passed in order to validate the process, it's kind of a CSRF check to avoid man in the middle attacks. Some don't recognise it or don't return it which will make the auth process invalid. Set this attribute to `False` in that case.

`REDIRECT_STATE` For those providers that don't recognise the `state` parameter, the app can add a `redirect_state` argument to the `redirect_uri` to mimic it. Set this value to `False` if the provider likes to verify the `redirect_uri` value and this parameter invalidates that check.

Example code:

```
from social_core.backends.oauth import BaseOAuth2

class GithubOAuth2(BaseOAuth2):
    """Github OAuth authentication backend"""
    name = 'github'
    AUTHORIZATION_URL = 'https://github.com/login/oauth/authorize'
    ACCESS_TOKEN_URL = 'https://github.com/login/oauth/access_token'
    SCOPE_SEPARATOR = ','
    EXTRA_DATA = [
        ('id', 'id'),
        ('expires', 'expires')
    ]

    def get_user_details(self, response):
        """Return user details from Github account"""
        return {'username': response.get('login'),
                'email': response.get('email') or '',
                'first_name': response.get('name')}

    def user_data(self, access_token, *args, **kwargs):
        """Loads user data from service"""
        url = 'https://api.github.com/user?' + urlencode({
            'access_token': access_token
        })
        return self.get_json(url)
```

OAuth1

OAuth1 process is a bit more trickier, Twitter Docs explains it quite well. Beside the `AUTHORIZATION_URL` and `ACCESS_TOKEN_URL` attributes, a third one is needed used when starting the process.

`REQUEST_TOKEN_URL = ''` During the auth process an unauthorized token is needed to start the process, later this token is exchanged for an `access_token`. This setting points to the API endpoint where that unauthorized token can be retrieved.

Example code:

```
from xml.dom import minidom

from social_core.backends.oauth import ConsumerBasedOAuth


class TripItOAuth(ConsumerBasedOAuth):
    """TripIt OAuth authentication backend"""
    name = 'tripit'
    AUTHORIZATION_URL = 'https://www.tripit.com/oauth/authorize'
    REQUEST_TOKEN_URL = 'https://api.tripit.com/oauth/request_token'
    ACCESS_TOKEN_URL = 'https://api.tripit.com/oauth/access_token'
    EXTRA_DATA = [('screen_name', 'screen_name')]

    def get_user_details(self, response):
        """Return user details from TripIt account"""
        try:
            first_name, last_name = response['name'].split(' ', 1)
        except ValueError:
            first_name = response['name']
            last_name = ''
```

```
        return {'username': response['screen_name'],
                'email': response['email'],
                'fullname': response['name'],
                'first_name': first_name,
                'last_name': last_name}

    def user_data(self, access_token, *args, **kwargs):
        """Return user data provided"""
        url = 'https://api.tripit.com/v1/get/profile'
        request = self.oauth_request(access_token, url)
        content = self.fetch_response(request)
        try:
            dom = minidom.parseString(content)
        except ValueError:
            return None

        return {
            'id': dom.getElementsByTagName('Profile')[0].getAttribute('ref'),
            'name': dom.getElementsByTagName(
                'public_display_name')[0].childNodes[0].data,
            'screen_name': dom.getElementsByTagName(
                'screen_name')[0].childNodes[0].data,
            'email': dom.getElementsByTagName(
                'is_primary')[0].parentNode.getElementsByTagName(
                'address')[0].childNodes[0].data,
        }
```

OpenId

OpenId is fair simpler that OAuth since it's used for authentication rather than authorization (regardless it's used for authorization too).

A single attribute is usually needed, the authentication URL endpoint.

`URL = ''` OpenId endpoint where to redirect the user.

Sometimes the URL is user dependant, like in myOpenId where the URL is `https://<user handler>.myopenid.com`. For those cases where the user must input it's handle (or full URL). The backend must override the `openid_url()` method to retrieve it and return a full URL to where the user will be redirected.

Example code:

```
from social_core.backends.open_id import OpenIdAuth
from social_core.exceptions import AuthMissingParameter


class LiveJournalOpenId(OpenIdAuth):
    """LiveJournal OpenID authentication backend"""
    name = 'livejournal'

    def get_user_details(self, response):
        """Generate username from identity url"""
        values = super(LiveJournalOpenId, self).get_user_details(response)
        values['username'] = values.get('username') or \
                             urlparse.urlsplit(response.identity_url)\
                                     .netloc.split('.', 1)[0]
        return values
```

```
    def openid_url(self):
        """Returns LiveJournal authentication URL"""
        if not self.data.get('openid_lj_user'):
            raise AuthMissingParameter(self, 'openid_lj_user')
        return 'http://%s.livejournal.com' % self.data['openid_lj_user']
```

Auth APIs

For others authentication types, a `BaseAuth` class is defined to help. Those custom auth methods must override the `auth_url()` and `auth_complete()` methods.

Example code:

```
from google.appengine.api import users

from social_core.backends.base import BaseAuth
from social_core.exceptions import AuthException


class GoogleAppEngineAuth(BaseAuth):
    """GoogleAppengine authentication backend"""
    name = 'google-appengine'

    def get_user_id(self, details, response):
        """Return current user id."""
        user = users.get_current_user()
        if user:
            return user.user_id()

    def get_user_details(self, response):
        """Return user basic information (id and email only)."""
        user = users.get_current_user()
        return {'username': user.user_id(),
                'email': user.email(),
                'fullname': '',
                'first_name': '',
                'last_name': ''}

    def auth_url(self):
        """Build and return complete URL."""
        return users.create_login_url(self.redirect_uri)

    def auth_complete(self, *args, **kwargs):
        """Completes login process, must return user instance."""
        if not users.get_current_user():
            raise AuthException('Authentication error')
        kwargs.update({'response': '', 'backend': self})
        return self.strategy.authenticate(*args, **kwargs)
```

9.2 Supported backends

Here's the list of currently supported backends.

9.2.1 Non-social backends

Email Auth

python-social-auth comes with an EmailAuth backend which comes handy when your site uses requires the plain old email and password authentication mechanism.

Actually that's a lie since the backend doesn't handle password at all, that's up to the developer to validate the password in and the proper place to do it is the pipeline, right after the user instance was retrieved or created.

The reason to leave password handling to the developer is because too many things are really tied to the project, like the field where the password is stored, salt handling, password hashing algorithm and validation. So just add the pipeline functions that will do that following the needs of your project.

Backend settings

SOCIAL_AUTH_EMAIL_FORM_URL = '/login-form/' Used to redirect the user to the login/signup form, it must have at least one field named `email`. Form submit should go to `/complete/email`, or if it goes to your view, then your view should complete the process calling `social_core.actions.do_complete`.

SOCIAL_AUTH_EMAIL_FORM_HTML = 'login_form.html' The template will be used to render the login/signup form to the user, it must have at least one field named `email`. Form submit should go to `/complete/email`, or if it goes to your view, then your view should complete the process calling `social_core.actions.do_complete`.

Email validation

Check *Email validation* pipeline in the pipeline docs.

Password handling

Here's an example of password handling to add to the pipeline:

```
def user_password(strategy, backend, user, is_new=False, *args, **kwargs):
    if backend.name != 'email':
        return

    password = strategy.request_data()['password']
    if is_new:
        user.set_password(password)
        user.save()
    elif not user.validate_password(password):
        # return {'user': None, 'social': None}
        raise AuthForbidden(backend)
```

Username Auth

python-social-auth comes with an UsernameAuth backend which comes handy when your site uses requires the plain old username and password authentication mechanism.

Actually that's a lie since the backend doesn't handle password at all, that's up to the developer to validate the password in and the proper place to do it is the pipeline, right after the user instance was retrieved or created.

The reason to leave password handling to the developer is because too many things are really tied to the project, like the field where the password is stored, salt handling, password hashing algorithm and validation. So just add the pipeline functions that will do that following the needs of your project.

Backend settings

SOCIAL_AUTH_USERNAME_FORM_URL = '/login-form/' Used to redirect the user to the login/signup form, it must have at least one field named `username`. Form submit should go to `/complete/username`, or if it goes to your view, then your view should complete the process calling `social_core.actions.do_complete`.

SOCIAL_AUTH_USERNAME_FORM_HTML = 'login_form.html' The template will be used to render the login/signup form to the user, it must have at least one field named `username`. Form submit should go to `/complete/username`, or if it goes to your view, then your view should complete the process calling `social_core.actions.do_complete`.

Password handling

Here's an example of password handling to add to the pipeline:

```
def user_password(strategy, user, is_new=False, *args, **kwargs):
    if strategy.backend.name != 'username':
        return

    password = strategy.request_data()['password']
    if is_new:
        user.set_password(password)
        user.save()
    elif not user.validate_password(password):
        # return {'user': None, 'social': None}
        raise AuthException(strategy.backend)
```

9.2.2 Base OAuth and OpenId classes

OAuth

OAuth communication demands a set of keys exchange to validate the client authenticity prior to user approbation. Twitter, and Facebook facilitates these keys by application registration, Google works the same, but provides the option for unregistered applications.

Check next sections for details.

OAuth backends also can store extra data in `UserSocialAuth.extra_data` field by defining a set of values names to retrieve from service response.

Settings is per backend and its name is dynamically checked using uppercase backend name as prefix:

```
SOCIAL_AUTH_<uppercase backend name>_EXTRA_DATA
```

Example:

```
SOCIAL_AUTH_FACEBOOK_EXTRA_DATA = [(..., ...)]
```

Settings must be a list of tuples mapping value name in response and value alias used to store. A third value (boolean) is supported, its purpose is to signal if the value should be discarded if it evaluates to `False`, this is to avoid replacing old (needed) values when they don't form part of current response. If not present, then this check is avoided and the value will replace any data.

OpenId

OpenId support is simpler to implement than OAuth. Google and Yahoo providers are supported by default, others are supported by POST method providing endpoint URL.

OpenId backends can store extra data in `UserSocialAuth.extra_data` field by defining a set of values names to retrieve from any of the used schemas, AttributeExchange and SimpleRegistration. As their keywords differ we need two settings.

Settings is per backend, so we have two possible values for each one. Name is dynamically checked using uppercase backend name as prefix:

```
SOCIAL_AUTH_<uppercase backend name>_SREG_EXTRA_DATA
SOCIAL_AUTH_<uppercase backend name>_AX_EXTRA_DATA
```

Example:

```
SOCIAL_AUTH_GOOGLE_SREG_EXTRA_DATA = [(..., ...)]
SOCIAL_AUTH_GOOGLE_AX_EXTRA_DATA = [(..., ...)]
```

Settings must be a list of tuples mapping value name in response and value alias used to store. A third value (boolean) is supported to, it's purpose is to signal if the value should be discarded if it evaluates to `False`, this is to avoid replacing old (needed) values when they don't form part of current response. If not present, then this check is avoided and the value will replace any data.

Username

The OpenId backend will check for a `username` key in the values returned by the server, but default to `first-name` + `last-name` if that key is missing. It's possible to indicate the username key in the values If the username is under a different key with a setting, but backends should have defined a default value. For example:

```
SOCIAL_AUTH_FEDORA_USERNAME_KEY = 'nickname'
```

This setting indicates that the username should be populated by the `nickname` value in the Fedora OpenId provider.

SAML

The SAML backend allows users to authenticate with any provider that supports the SAML 2.0 protocol (commonly used for corporate or academic single sign on).

The SAML backend for python-social-auth allows your web app to act as a SAML Service Provider. You can configure one or more SAML Identity Providers that users can use for authentication. For example, if your users are students, you could enable Harvard and MIT as identity providers, so that students of either of those two universities can use their campus login to access your app.

Required Dependency

You must install python-saml 2.2.0 or higher in order to use this backend, if using Python 3, you need to install python3-saml 1.2.1 or higher.

Required Configuration

At a minimum, you must add the following to your project's settings:

- `SOCIAL_AUTH_SAML_SP_ENTITY_ID`: The SAML Entity ID for your app. This should be a URL that includes a domain name you own. It doesn't matter what the URL points to. Example: `http://saml.yoursite.com`
- `SOCIAL_AUTH_SAML_SP_PUBLIC_CERT`: The X.509 certificate string for the key pair that your app will use. You can generate a new self-signed key pair with:

```
openssl req -new -x509 -days 3652 -nodes -out saml.crt -keyout saml.key
```

  The contents of `saml.crt` should then be used as the value of this setting (you can omit the first and last lines, which aren't required).

- `SOCIAL_AUTH_SAML_SP_PRIVATE_KEY`: The private key to be used by your app. If you used the example openssl command given above, set this to the contents of `saml.key` (again, you can omit the first and last lines).
- `SOCIAL_AUTH_SAML_ORG_INFO`: A dictionary that contains information about your app. You must specify values for English at a minimum. Each language's entry should specify a `name` (not shown to the user), a `displayname` (shown to the user), and a URL. See the following example:

```
{
    "en-US": {
        "name": "example",
        "displayname": "Example Inc.",
        "url": "http://example.com",
    }
}
```

- `SOCIAL_AUTH_SAML_TECHNICAL_CONTACT`: A dictionary with two values, `givenName` and `emailAddress`, describing the name and email of a technical contact responsible for your app. Example:

```
{"givenName": "Tech Gal", "emailAddress": "technical@example.com"}
```

- `SOCIAL_AUTH_SAML_TECHNICAL_CONTACT`: A dictionary with two values, `givenName` and `emailAddress`, describing the name and email of a support contact for your app. Example:

```
SOCIAL_AUTH_SAML_SUPPORT_CONTACT = {
    "givenName": "Support Guy",
    "emailAddress": "support@example.com",
}
```

- `SOCIAL_AUTH_SAML_ENABLED_IDPS`: The most important setting. List the Entity ID, SSO URL, and x.509 public key certificate for each provider that your app wants to support. The SSO URL must support the `HTTP-Redirect` binding. You can get these values from the provider's XML metadata. Here's an example, for TestShib (the values come from TestShib's metadata):

```
{
    "testshib": {
        "entity_id": "https://idp.testshib.org/idp/shibboleth",
        "url": "https://idp.testshib.org/idp/profile/SAML2/Redirect/SSO",
        "x509cert": "MIIEDjCCAvagAwIBAgIBADA ... 8Bbnl+ev0peYzxFyF5sQA==",
    }
}
```

Basic Usage

- Set all of the required configuration variables described above.
- Generate the SAML XML metadata for your app. The best way to do this is to create a new view/page/URL in your app that will call the backend's `generate_metadata_xml()` method. Here's an example of how to do this in Django:

```
def saml_metadata_view(request):
    complete_url = reverse('social:complete', args=("saml", ))
    saml_backend = load_backend(
        load_strategy(request),
        "saml",
        redirect_uri=complete_url,
    )
    metadata, errors = saml_backend.generate_metadata_xml()
    if not errors:
        return HttpResponse(content=metadata, content_type='text/xml')
```

- Download the metadata for your app that was generated by the above method, and send it to each Identity Provider (IdP) that you wish to use. Each IdP must install and configure your metadata on their system before it will work.
- Now everything is set! To allow users to login with any given IdP, you need to give them a link to the python-social-auth "begin"/"auth" URL and include an `idp` query parameter that specifies the name of the IdP to use. This is needed since the backend supports multiple IdPs. The names of the IdPs are the keys used in the `SOCIAL_AUTH_SAML_ENABLED_IDPS` setting.

  Django example:

```
# In view:
context['testshib_url'] = u"{base}?{params}".format(
    base=reverse('social:begin', kwargs={'backend': 'saml'}),
    params=urllib.urlencode({'next': '/home', 'idp': 'testshib'})
)
# In template:
<a href="{{ testshib_url }}">TestShib Login</a>
# Result:
<a href="/login/saml/?next=%2Fhome&amp;idp=testshib">TestShib Login</a>
```

- Testing with the TestShib provider is recommended, as it is known to work well.

Advanced Settings

- `SOCIAL_AUTH_SAML_SP_EXTRA`: This can be set to a dict, and any key/value pairs specified here will be passed to the underlying `python-saml` library configuration's `sp` setting. Refer to the `python-saml` documentation for details.

- `SOCIAL_AUTH_SAML_SECURITY_CONFIG`: This can be set to a dict, and any key/value pairs specified here will be passed to the underlying `python-saml` library configuration's `security` setting. Two useful keys that you can set are `metadataCacheDuration` and `metadataValidUntil`, which control the expiry time of your XML metadata. By default, a cache duration of 10 days will be used, which means that IdPs are allowed to cache your metadata for up to 10 days, but no longer. `metadataCacheDuration` must be specified as an ISO 8601 duration string (e.g. *P1D* for one day).

Advanced Usage

You can subclass the `SAMLAuth` backend to provide custom functionality. In particular, there are two methods that are designed for subclasses to override:

- `get_idp(self, idp_name)`: Given the name of an IdP, return an instance of `SAMLIdentityProvider` with the details of the IdP. Override this method if you wish to use some other method for configuring the available identity providers, such as fetching them at runtime from another server, or using a list of providers from a Shibboleth federation.
- `_check_entitlements(self, idp, attributes)`: This method gets called during the login process and is where you can decide to accept or reject a user based on the user's SAML attributes. For example, you can restrict access to your application to only accept users who belong to a certain department. After inspecting the passed attributes parameter, do nothing to allow the user to login, or raise `social_core.exceptions.AuthForbidden` to reject the user.

9.2.3 Social backends

Amazon

Amazon implemented OAuth2 protocol for their authentication mechanism. To enable `python-social-auth` support follow this steps:

1. Go to Amazon App Console and create an application.
2. Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_AMAZON_KEY = '...'
SOCIAL_AUTH_AMAZON_SECRET = '...'
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.amazon.AmazonOAuth2',
    ...
)
```

Further documentation at Website Developer Guide and Getting Started for Web.

Note: This backend supports TLSv1 protocol since SSL will be deprecated from May 25, 2015

Angel List

Angel uses OAuth v2 for Authentication.

- Register a new application at the Angel List API, and

- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_ANGEL_KEY = ''
SOCIAL_AUTH_ANGEL_SECRET = ''
```

- extra scopes can be defined by using:

```
SOCIAL_AUTH_ANGEL_AUTH_EXTRA_ARGUMENTS = {'scope': 'email messages'}
```

Note: Angel List does not currently support returning `state` parameter used to validate the auth process.

AOL

AOL OpenId doesn't require major settings beside being defined on `AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.aol.AOLOpenId',
    ...
)
```

Appsfuel

Appsfuel uses OAuth v2 for Authentication check the official docs too.

- Sign up at the Appsfuel Developer Program
- Create and verify a new app
- On the dashboard click on **Show API keys**
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_APPSFUEL_KEY = '<App UID>'
SOCIAL_AUTH_APPSFUEL_SECRET = '<App secret>'
```

Appsfuel gives you the chance to integrate with **Live** or **Sandbox** env.

Appsfuel Live

- Add 'social_core.backends.contrib.appsfuel.AppsfuelBackend' into your `AUTHENTICATION_BACKENDS`.
- Then you can start using `{% url social:begin 'appsfuel' %}` in your templates

Appsfuel Sandbox

- Add `'social_core.backends.appsfuel.AppsfuelOAuth2Sandbox'` into your `AUTHENTICATION_BACKENDS`.
- Then you can start using `{% url social:begin 'appsfuel-sandbox' %}` in your templates
- Define the settings:

```
SOCIAL_AUTH_APPSFUEL_SANDBOX_KEY = '<App UID>'
SOCIAL_AUTH_APPSFUEL_SANDBOX_SECRET = '<App secret>'
```

ArcGIS

ArcGIS uses OAuth2 for authentication.

- Register a new application at ArcGIS Developer Center.

OAuth2

1. Add the OAuth2 backend to your settings page:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.arcgis.ArcGISOAuth2',
    ...
)
```

2. Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_ARCGIS_KEY = ''
SOCIAL_AUTH_ARCGIS_SECRET = ''
```

Microsoft Azure Active Directory

To enable OAuth2 support:

- Fill in `Client ID` and `Client Secret` settings. These values can be obtained easily as described in Azure AD Application Registration doc:

```
SOCIAL_AUTH_AZUREAD_OAUTH2_KEY = ''
SOCIAL_AUTH_AZUREAD_OAUTH2_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_AZUREAD_OAUTH2_RESOURCE = ''
```

  This is the resource you would like to access after authentication succeeds. Some of the possible values are: `https://graph.windows.net` or `https://<your Sharepoint site name>-my.sharepoint.com`.

Battle.net

Blizzard implemented OAuth2 protocol for their authentication mechanism. To enable `python-social-auth` support follow this steps:

1. Go to Battlenet Developer Portal and create an application.
2. Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_BATTLENET_OAUTH2_KEY = '...'
SOCIAL_AUTH_BATTLENET_OAUTH2_SECRET = '...'
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.battlenet.BattleNetOAuth2',
    ...
)
```

Note: If you want to allow the user to choose a username from his own characters, some further steps are required, see the use cases part of the documentation. To get the account id and battletag use the user_data function, as account id is no longer passed inherently.

Another note: If you get a 500 response "Internal Server Error" the API now requires https on callback endpoints.

Further documentation at Developer Guide.

Beats

Beats supports OAuth 2.

- Register a new application at Beats Music API, and follow the instructions below.

OAuth2

Add the Beats OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.beats.BeatsOAuth2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_BEATS_OAUTH2_KEY = ''
SOCIAL_AUTH_BEATS_OAUTH2_SECRET = ''
```

Behance

DEPRECATED NOTICE

NOTE: IT SEEMS THAT BEHANCE HAS DROPPED THEIR OAUTH2 SUPPORT WITHOUT MUCH NOTICE BESIDE A BLOG POST ON SEPTEMBER 2014 MENTIONING THAT IT WILL BE INTRODUCED "SOON". THIS BACKEND IS IN DEPRECATED STATE FOR NOW.

Behance uses OAuth2 for its auth mechanism.

- Register a new application at Behance App Registration, set your application name, website and redirect URI.
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_BEHANCE_KEY = ''
SOCIAL_AUTH_BEHANCE_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_BEHANCE_SCOPE = [...]
```

Check available permissions at Possible Scopes. Also check the rest of their doc at Behance Developer Documentation.

Belgium EID

Belgium EID OpenId doesn't require major settings beside being defined on `AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.belgiumeid.BelgiumEIDOpenId',
    ...
)
```

Bitbucket

Bitbucket supports both OAuth2 and OAuth1 logins.

1. Register a new OAuth Consumer by following the instructions in the Bitbucket documentation: OAuth on Bitbucket

   Note: For OAuth2, your consumer MUST have the "account" scope otherwise the user profile information (username, name, etc.) won't be accessible.

2. Configure the appropriate settings for OAuth2 or OAuth1 (see below).

OAuth2

- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_BITBUCKET_OAUTH2_KEY = '<your-consumer-key>'
SOCIAL_AUTH_BITBUCKET_OAUTH2_SECRET = '<your-consumer-secret>'
```

- If you would like to restrict access to only users with verified e-mail addresses, set `SOCIAL_AUTH_BITBUCKET_OAUTH2_VERIFIED_EMAILS_ONLY = True` By default the setting is set to `False` since it's possible for a project to gather this information by other methods.

OAuth1

- OAuth1 works similarly to OAuth2, but you must fill in the following settings instead:

```
SOCIAL_AUTH_BITBUCKET_KEY = '<your-consumer-key>'
SOCIAL_AUTH_BITBUCKET_SECRET = '<your-consumer-secret>'
```

- If you would like to restrict access to only users with verified e-mail addresses, set `SOCIAL_AUTH_BITBUCKET_VERIFIED_EMAILS_ONLY = True`. By default the setting is set to `False` since it's possible for a project to gather this information by other methods.

User ID

Bitbucket recommends the use of UUID as the user identifier instead of `username` since they can change and impose a security risk. For that reason `UUID` is used by default, but for backward compatibility reasons, it's possible to get the old behavior again by defining this setting:

```
SOCIAL_AUTH_BITBUCKET_USERNAME_AS_ID = True
```

Box.net

Box works similar to Facebook (OAuth2).

- Register an application at Manage Box Applications
- Fill the **Consumer Key** and **Consumer Secret** values in your settings:

```
SOCIAL_AUTH_BOX_KEY = ''
SOCIAL_AUTH_BOX_SECRET = ''
```

- By default the token is not permanent, it will last an hour. To refresh the access token just do:

```
from social_django.utils import load_strategy

strategy = load_strategy(backend='box')
user = User.objects.get(pk=foo)
social = user.social_auth.filter(provider='box')[0]
social.refresh_token(strategy=strategy)
```

ChangeTip

ChangeTip

- Register a new application at ChangeTip, set the callback URL to `http://example.com/complete/changetip/` replacing `example.com` with your domain.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_CHANGETIP_KEY = ''
SOCIAL_AUTH_CHANGETIP_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_CHANGETIP_SCOPE = [...]
```

  See auth scopes at ChangeTip OAuth docs.

Clef

Clef works similar to Facebook (OAuth).

- Register a new application at Clef Developers, set the callback URL to `http://example.com/complete/clef/` replacing `example.com` with your domain.
- Fill `App Id` and `App Secret` values in the settings:

```
SOCIAL_AUTH_CLEF_KEY = ''
SOCIAL_AUTH_CLEF_SECRET = ''
```

Coinbase

Coinbase uses OAuth2.

- Register an application at Coinbase
- Fill in the **Client Id** and **Client Secret** values in your settings:

```
SOCIAL_AUTH_COINBASE_KEY = ''
SOCIAL_AUTH_COINBASE_SECRET = ''
```

- Set the `redirect_url` on coinbase. Make sure to include the trailing slash, eg. `http://hostname/complete/coinbase/`
- Specify scopes with:

```
SOCIAL_AUTH_COINBASE_SCOPE = [...]
```

  By default the scope is set to `balance`.

Coursera

Coursera uses a variant of OAuth2 authentication. The details of the API can be found at OAuth2-based APIs - Coursera Technology.

Take the following steps in order to use the backend:

1. Create an account at Coursera.
2. Open Developer Console, create an organisation and application.

3. Set **Client ID** as a `SOCIAL_AUTH_COURSERA_KEY` and **Secret Key** as a `SOCIAL_AUTH_COURSERA_SECRET` in your local settings.

4. Add the backend to `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.coursera.CourseraOAuth2',
    ...
)
```

DailyMotion

DailyMotion uses OAuth2. In order to enable the backend follow:

- Register an application at DailyMotion Developer Portal
- Fill in the **Client Id** and **Client Secret** values in your settings:

```
SOCIAL_AUTH_DAILYMOTION_KEY = ''
SOCIAL_AUTH_DAILYMOTION_SECRET = ''
```

- Set the `Callback URL` to `http://<your hostname>/complete/dailymotion/`

- Specify scopes with:

```
SOCIAL_AUTH_DAILYMOTION_SCOPE = [...]
```

  Available scopes are listed in the Requesting Extended Permissions section.

DigitalOcean

DigitalOcean uses OAuth2 for its auth process. See the full DigitalOcean developer's documentation for more information.

- Register a new application in the Apps & API page in the DigitalOcean control panel, setting the callback URL to `http://example.com/complete/digitalocean/` replacing `example.com` with your domain.
- Fill the `Client ID` and `Client Secret` values from GitHub in the settings:

```
SOCIAL_AUTH_DIGITALOCEAN_KEY = ''
SOCIAL_AUTH_DIGITALOCEAN_SECRET = ''
```

- By default, only `read` permissions are granted. In order to create, destroy, and take other actions on the user's resources, you must request `read write` permissions like so:

```
SOCIAL_AUTH_DIGITALOCEAN_AUTH_EXTRA_ARGUMENTS = {'scope': 'read write'}
```

Disqus

Disqus uses OAuth v2 for Authentication.

- Register a new application at the Disqus API, and
- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_DISQUS_KEY = ''
SOCIAL_AUTH_DISQUS_SECRET = ''
```

- extra scopes can be defined by using:

```
SOCIAL_AUTH_DISQUS_AUTH_EXTRA_ARGUMENTS = {'scope': 'likes comments relationships
↪'}
```

  Check Disqus Auth API for details.

Docker

Docker.io OAuth2

Docker.io now supports OAuth2 for their API. In order to set it up:

- Register a new application by following the instructions in their website: Register Your Application
- Fill **Consumer Key** and **Consumer Secret** values in settings:

```
SOCIAL_AUTH_DOCKER_KEY = ''
SOCIAL_AUTH_DOCKER_SECRET = ''
```

- Add `'social_core.backends.docker.DockerOAuth2'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.

Douban

Douban supports OAuth 1 and 2.

Douban OAuth1

Douban OAuth 1 works similar to Twitter OAuth.

Douban offers per application keys named `Consumer Key` and `Consumer Secret`. To enable Douban OAuth these two keys are needed. Further documentation at Douban Services & API:

- Register a new application at Douban API Key, make sure to mark the **web application** checkbox.
- Fill **Consumer Key** and **Consumer Secret** values in settings:

```
SOCIAL_AUTH_DOUBAN_KEY = ''
SOCIAL_AUTH_DOUBAN_SECRET = ''
```

- Add `'social_core.backends.douban.DoubanOAuth'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.

Douban OAuth2

Recently Douban launched their OAuth2 support and the new developer site, you can find documentation at Douban Developers. To setup OAuth2 follow:

- Register a new application at Create A Douban App, make sure to mark the **web application** checkbox.
- Fill **Consumer Key** and **Consumer Secret** values in settings:

```
SOCIAL_AUTH_DOUBAN_OAUTH2_KEY = ''
SOCIAL_AUTH_DOUBAN_OAUTH2_SECRET = ''
```

- Add `'social_core.backends.douban.DoubanOAuth2'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.

Dribbble

Dribbble

- Register a new application at Dribbble, set the callback URL to `http://example.com/complete/dribbble/` replacing `example.com` with your domain.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_DRIBBBLE_KEY = ''
SOCIAL_AUTH_DRIBBBLE_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_DRIBBBLE_SCOPE = [...]
```

  See auth scopes at Dribbble Developer docs.

Drip

Drip uses OAuth v2 for Authentication.

- Register a new application with Drip, and
- fill `Client ID` and `Client Secret` from getdrip.com values in the settings:

```
SOCIAL_AUTH_DRIP_KEY = ''
SOCIAL_AUTH_DRIP_SECRET = ''
```

Dropbox

Dropbox supports both OAuth 1 and 2.

- Register a new application at Dropbox Developers, and follow the instructions below for the version of OAuth for which you are adding support.

OAuth2 Api V2

Add the Dropbox OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social.backends.dropbox.DropboxOAuth2V2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_DROPBOX_OAUTH2_KEY = ''
SOCIAL_AUTH_DROPBOX_OAUTH2_SECRET = ''
```

OAuth1

Deprecated since version V1: api is deprecated.

https://blogs.dropbox.com/developers/2016/06/api-v1-deprecated/ Add the Dropbox OAuth backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.dropbox.DropboxOAuth',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_DROPBOX_KEY = ''
SOCIAL_AUTH_DROPBOX_SECRET = ''
```

OAuth2

Deprecated since version V1: api is deprecated.

https://blogs.dropbox.com/developers/2016/06/api-v1-deprecated/

Add the Dropbox OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.dropbox.DropboxOAuth2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_DROPBOX_OAUTH2_KEY = ''
SOCIAL_AUTH_DROPBOX_OAUTH2_SECRET = ''
```

Edmodo

Edmodo supports OAuth 2.

- Register a new application at Edmodo Connect API, and follow the instructions below.
- Add the Edmodo OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.edmodo.EdmodoOAuth2',
    ...
)
```

- Fill `App Key`, `App Secret` and `App Scope` values in the settings:

```
SOCIAL_AUTH_EDMODO_OAUTH2_KEY = ''
SOCIAL_AUTH_EDMODO_OAUTH2_SECRET = ''
SOCIAL_AUTH_EDMODO_SCOPE = ['basic']
```

EVE Online Single Sign-On (SSO)

The EVE Single Sign-On (SSO) works similar to GitHub (OAuth2).

- Register a new application at EVE Developers, set the callback URL to `http://example.com/complete/eveonline/` replacing `example.com` with your domain.
- Fill the `Client ID` and `Secret Key` values from EVE Developers in the settings:

```
SOCIAL_AUTH_EVEONLINE_KEY = ''
SOCIAL_AUTH_EVEONLINE_SECRET = ''
```

- If you want to use EVE Character names as user names, use this setting:

```
SOCIAL_AUTH_CLEAN_USERNAMES = False
```

- If you want to access EVE Online's CREST API, use:

```
SOCIAL_AUTH_EVEONLINE_SCOPE = ['publicData']
```

Evernote OAuth

Evernote OAuth 1.0 for its authentication workflow.

- Register a new application at Evernote API Key form.
- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_EVERNOTE_KEY = ''
SOCIAL_AUTH_EVERNOTE_SECRET = ''
```

Sandbox

Evernote supports a sandbox mode for testing, there's a custom backend for it which name is `evernote-sandbox` instead of `evernote`. Same settings apply but use these instead:

```
SOCIAL_AUTH_EVERNOTE_SANDBOX_KEY = ''
SOCIAL_AUTH_EVERNOTE_SANDBOX_SECRET = ''
```

Facebook

OAuth2

Facebook uses OAuth2 for its auth process. Further documentation at Facebook development resources:

- Register a new application at Facebook App Creation, don't use `localhost` as `App Domains` and `Site URL` since Facebook won't allow them. Use a placeholder like `myapp.com` and define that domain in your `/etc/hosts` or similar file.
- fill `App Id` and `App Secret` values in values:

```
SOCIAL_AUTH_FACEBOOK_KEY = ''
SOCIAL_AUTH_FACEBOOK_SECRET = ''
```

- Define `SOCIAL_AUTH_FACEBOOK_SCOPE` to get extra permissions from facebook. Email is not sent by default, to get it, you must request the `email` permission:

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

- Define `SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS` to pass extra parameters to https://graph.facebook.com/me when gathering the user profile data (you need to explicitly ask for fields like `email` using `fields` key):

```
SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS = {
  'locale': 'ru_RU',
  'fields': 'id, name, email, age_range'
}
```

If you define a redirect URL in Facebook setup page, be sure to not define http://127.0.0.1:8000 or http://localhost:8000 because it won't work when testing. Instead I define http://myapp.com and setup a mapping on `/etc/hosts`.

Currently the backend uses Facebook API version *2.8*, this value can be overridden by the following setting if needed:

```
SOCIAL_AUTH_FACEBOOK_API_VERSION = '2.9'
```

Canvas Application

If you need to perform authentication from Facebook Canvas application:

- Create your canvas application at http://developers.facebook.com/apps
- In Facebook application settings specify your canvas URL `mysite.com/fb` (current default)
- Setup your Python Social Auth settings and your application namespace:

```
SOCIAL_AUTH_FACEBOOK_APP_KEY = ''
SOCIAL_AUTH_FACEBOOK_APP_SECRET = ''
SOCIAL_AUTH_FACEBOOK_APP_NAMESPACE = ''
```

- Launch your testing server on port 80 (use sudo or nginx or apache) for browser to be able to load it when Facebook calls canvas URL
- Open your Facebook page via http://apps.facebook.com/app_namespace or better via http://www.facebook.com/pages/user-name/user-id?sk=app_app-id
- After that you will see this page in a right way and will able to connect to application and login automatically after connection
- Provide a template to be rendered, it must have this JavaScript snippet (or similar) in it:

```
<script type="text/javascript">
    var domain = 'https://apps.facebook.com/',
        redirectURI = domain + {{ FACEBOOK_APP_NAMESPACE }} + '/';

    window.top.location = 'https://www.facebook.com/dialog/oauth/' +
                          '?client_id={{ FACEBOOK_KEY }}' +
                          '&redirect_uri=' +␣
↪encodeURIComponent(redirectURI) +
                          '&scope={{ FACEBOOK_EXTENDED_PERMISSIONS }}';
</script>
```

More info on the topic at Facebook Canvas Application Authentication.

Fedora

Fedora OpenId doesn't require major settings beside being defined on `AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.fedora.FedoraOpenId',
    ...
)
```

Fitbit

Fitbit supports both OAuth 2.0 and OAuth 1.0a logins. OAuth 2 is preferred for new integrations, as OAuth 1.0a does not support getting heartrate or location and will be deprecated in the future.

1. Register a new OAuth Consumer here

2. Configure the appropriate settings for OAuth 2.0 or OAuth 1.0a (see below).

OAuth 2.0 or OAuth 1.0a

- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_FITBIT_KEY = '<your-consumer-key>'
SOCIAL_AUTH_FITBIT_SECRET = '<your-consumer-secret>'
```

OAuth 2.0 specific settings

By default, only the `profile` scope is requested. To request more scopes, set SOCIAL_AUTH_FITBIT_SCOPE:

```
SOCIAL_AUTH_FITBIT_SCOPE = [
  'activity',
  'heartrate',
  'location',
  'nutrition',
  'profile',
  'settings',
  'sleep',
  'social',
  'weight'
]
```

The above will request all permissions from the user.

Flickr

Flickr uses OAuth v1.0 for authentication.

- Register a new application at the Flickr App Garden, and
- fill `Key` and `Secret` values in the settings:

```
SOCIAL_AUTH_FLICKR_KEY = ''
SOCIAL_AUTH_FLICKR_SECRET = ''
```

- Flickr might show a messages saying "Oops! Flickr doesn't recognise the permission set.", if encountered with this error, just define this setting:

```
SOCIAL_AUTH_FLICKR_AUTH_EXTRA_ARGUMENTS = {'perms': 'read'}
```

Foursquare

Foursquare uses OAuth2. In order to enable the backend follow:

- Register an application at Foursquare Developers Portal, set the `Redirect URI` to `http://<your hostname>/complete/foursquare/`
- Fill in the **Client Id** and **Client Secret** values in your settings:

```
SOCIAL_AUTH_FOURSQUARE_KEY = ''
SOCIAL_AUTH_FOURSQUARE_SECRET = ''
```

GitHub

GitHub works similar to Facebook (OAuth).

- Register a new application at GitHub Developers, set the callback URL to `http://example.com/complete/github/` replacing `example.com` with your domain.
- Fill the `Client ID` and `Client Secret` values from GitHub in the settings:

```
SOCIAL_AUTH_GITHUB_KEY = ''
SOCIAL_AUTH_GITHUB_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_GITHUB_SCOPE = [...]
```

GitHub for Organizations

When defining authentication for organizations, use the `GithubOrganizationOAuth2` backend instead. The settings are the same as the non-organization backend, but the names must be:

```
SOCIAL_AUTH_GITHUB_ORG_*
```

Be sure to define the organization name using the setting:

```
SOCIAL_AUTH_GITHUB_ORG_NAME = ''
```

This name will be used to check that the user really belongs to the given organization and discard it if they're not part of it.

GitHub for Teams

Similar to `GitHub for Organizations`, there's a GitHub for Teams backend, use the backend `GithubTeamOAuth2`. The settings are the same as the basic backend, but the names must be:

```
SOCIAL_AUTH_GITHUB_TEAM_*
```

Be sure to define the `Team ID` using the setting:

```
SOCIAL_AUTH_GITHUB_TEAM_ID = ''
```

This `id` will be used to check that the user really belongs to the given team and discard it if they're not part of it.

Github for Enterprises

Check the docs *GitHub Enterprise* if planning to use Github Enterprises.

GitHub Enterprise

GitHub Enterprise works similar to regular Github, which is in turn based on Facebook (OAuth).

- Register a new application on your instance of GitHub Enterprise Developers, set the callback URL to `http://example.com/complete/github/` replacing `example.com` with your domain.

- Set the API URL for your Github Enterprise appliance:

  SOCIAL_AUTH_GITHUB_ENTERPRISE_API_URL = 'https://git.example.com/api/v3/'

- Fill the `Client ID` and `Client Secret` values from GitHub in the settings:

  SOCIAL_AUTH_GITHUB_ENTERPRISE_KEY = '' SOCIAL_AUTH_GITHUB_ENTERPRISE_SECRET = ''

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_GITHUB_ENTERPRISE_SCOPE = [...]
```

GitHub Enterprise for Organizations

When defining authentication for organizations, use the `GithubEnterpriseOrganizationOAuth2` backend instead. The settings are the same as the non-organization backend, but the names must be:

```
SOCIAL_AUTH_GITHUB_ENTERPRISE_ORG_*
```

Be sure to define the organization name using the setting:

```
SOCIAL_AUTH_GITHUB_ENTERPRISE_ORG_NAME = ''
```

This name will be used to check that the user really belongs to the given organization and discard it if they're not part of it.

GitHub Enterprise for Teams

Similar to `GitHub Enterprise for Organizations`, there's a GitHub for Teams backend, use the backend `GithubEnterpriseTeamOAuth2`. The settings are the same as the basic backend, but the names must be:

```
SOCIAL_AUTH_GITHUB_ENTERPRISE_TEAM_*
```

Be sure to define the `Team ID` using the setting:

```
SOCIAL_AUTH_GITHUB_ENTERPRISE_TEAM_ID = ''
```

This `id` will be used to check that the user really belongs to the given team and discard it if they're not part of it.

GitLab

GitLab supports OAuth2 protocol.

- Register a new application at GitLab Applications.
- Set the callback URL to `http://example.com/complete/gitlab/` replacing `example.com` with your domain. Drop the trailing slash if the project doesn't use it, the URL **must** match the value sent.
- Ensure to mark the `read_user` scope. If marking `api` scope too, define:

```
SOCIAL_AUTH_GITLAB_SCOPE = ['api']
```

- Fill the `Client ID` and `Client Secret` values from GitLab in the settings:

```
SOCIAL_AUTH_GITLAB_KEY = ''
SOCIAL_AUTH_GITLAB_SECRET = ''
```

If your GitLab setup resides in another domain, then add the following setting:

```
SOCIAL_AUTH_GITLAB_API_URL = 'https://example.com'
```

it must be the **full url** to your GitLab setup.

Google

This section describes how to setup the different services provided by Google.

Google OAuth

Attention: Google OAuth deprecation Important: OAuth 1.0 was officially deprecated on April 20, 2012, and will be shut down on April 20, 2015. We encourage you to migrate to any of the other protocols.

Google provides `Consumer Key` and `Consumer Secret` keys to registered applications, but also allows unregistered application to use their authorization system with, but beware that this method will display a security banner to the user telling that the application is not trusted.

Check Google OAuth and make your choice.

- fill `Consumer Key` and `Consumer Secret` values:

```
SOCIAL_AUTH_GOOGLE_OAUTH_KEY = ''
SOCIAL_AUTH_GOOGLE_OAUTH_SECRET = ''
```

anonymous values will be used if not configured as described in their OAuth reference

- setup any needed extra scope in:

```
SOCIAL_AUTH_GOOGLE_OAUTH_SCOPE = [...]
```

Google OAuth2

Recently Google launched OAuth2 support following the definition at *OAuth2 draft*. It works in a similar way to plain OAuth mechanism, but developers **must** register an application and apply for a set of keys. Check Google OAuth2 document for details.

When creating the application in the Google Console be sure to fill the `PRODUCT NAME` at `API & auth -> Consent screen` form.

To enable OAuth2 support:

- fill `Client ID` and `Client Secret` settings, these values can be obtained easily as described on OAuth2 Registering doc:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = ''
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = ''
```

- setup any needed extra scope:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE = [...]
```

Check which applications can be included in their Google Data Protocol Directory

Google+ Sign-In

Google+ Sign In works a lot like OAuth2, but most of the initial work is done by their Javascript which thens calls a defined handler to complete the auth process.

- To enable the backend create an application using the Google console and following the steps from the official guide. Make sure to enable the Google+ API in the console.
- Fill in the key settings looking inside the Google console the subsection `Credentials` inside `API & auth`:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.google.GooglePlusAuth',
)

SOCIAL_AUTH_GOOGLE_PLUS_KEY = '...'
SOCIAL_AUTH_GOOGLE_PLUS_SECRET = '...'
```

`SOCIAL_AUTH_GOOGLE_PLUS_KEY` corresponds to the variable `CLIENT ID`. `SOCIAL_AUTH_GOOGLE_PLUS_SECRET` corresponds to the variable `CLIENT SECRET`.

- Add the sign-in button to your template, you can use the SDK button or add your own and attach the click handler to it (check Google+ Identity Sign-In documentation about it):

```
<div id="google-plus-button">Google+ Sign In</div>
```

- Add the Javascript snippet in the same template as above:

```
<script src="https://apis.google.com/js/api:client.js"></script>

<script type="text/javascript">
  gapi.load('auth2', function () {
    var auth2;

    auth2 = gapi.auth2.init({
      client_id: "<PUT SOCIAL_AUTH_GOOGLE_PLUS_KEY HERE>",
      scope: "<PUT BACKEND SCOPE HERE>"
    });

    auth2.then(function () {
      var button = document.getElementById("google-plus-button");
      console.log("User is signed-in in Google+ platform?", auth2.isSignedIn.
↪get() ? "Yes" : "No");

      auth2.attachClickHandler(button, {}, function (googleUser) {
        // Send access-token to backend to finish the authenticate
        // with your application

        var authResponse = googleUser.getAuthResponse();
        var $form;
        var $input;
```

```
        $form = $("<form>");
        $form.attr("action", "/complete/google-plus");
        $form.attr("method", "post");
        $input = $("<input>");
        $input.attr("name", "access_token");
        $input.attr("value", authResponse.access_token);
        $form.append($input);
        // Add csrf-token if needed
        $(document.body).append($form);
        $form.submit();
      });
    });
  });
</script>
```

- Logging out

  Logging-out can be tricky when using the the platform SDK because it can trigger an automatic sign-in when listening to the user status change. With the method show above, that won't happen, but if the UI depends more in the SDK values than the backend, then things can get out of sync easilly. To prevent this, the user should be logged-out from Google+ platform too. This can be accomplished by doing:

```
<script type="text/javascript">
  gapi.load('auth2', function () {
    var auth2;

    auth2 = gapi.auth2.init({
      client_id: "{{ plus_id }}",
      scope: "{{ plus_scope }}"
    });

    auth2.then(function () {
      if (auth2.isSignedIn.get()) {
        $('#logout').on('click', function (event) {
          event.preventDefault();
          auth2.signOut().then(function () {
            console.log("Logged out from Google+ platform");
            document.location = "/logout";
          });
        });
      }
    });
  });
</script>
```

Google OpenId

Google OpenId works straightforward, not settings are needed. Domains or emails whitelists can be applied too, check the whitelists settings for details.

Orkut

As of September 30, 2014, Orkut has been shut down.

User identification

Optional support for static and unique Google Profile ID identifiers instead of using the e-mail address for account association can be enabled with:

```
SOCIAL_AUTH_GOOGLE_OAUTH_USE_UNIQUE_USER_ID = True
```

or:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_USE_UNIQUE_USER_ID = True
```

depending on the backends in use.

Refresh Tokens

To get an OAuth2 refresh token along with the access token, you must pass an extra argument: `access_type=offline`. To do this with Google+ sign-in:

```
SOCIAL_AUTH_GOOGLE_PLUS_AUTH_EXTRA_ARGUMENTS = {
      'access_type': 'offline'
}
```

Scopes deprecation

Google is deprecating the full-url scopes from Sept 1, 2014 in favor of `Google+ API` and the recently introduced shorter scopes names. But `python-social-auth` already introduced the scopes change at e3525187 which was released at `v0.1.24`.

But, to enable the new scopes the application requires `Google+ API` to be enabled in the Google console dashboard, the change is quick and quite simple, but if any developer desires to keep using the old scopes, it's possible with the following settings:

```
# Google OAuth2 (google-oauth2)
SOCIAL_AUTH_GOOGLE_OAUTH2_IGNORE_DEFAULT_SCOPE = True
SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE = [
    'https://www.googleapis.com/auth/userinfo.email',
    'https://www.googleapis.com/auth/userinfo.profile'
]

# Google+ SignIn (google-plus)
SOCIAL_AUTH_GOOGLE_PLUS_IGNORE_DEFAULT_SCOPE = True
SOCIAL_AUTH_GOOGLE_PLUS_SCOPE = [
    'https://www.googleapis.com/auth/plus.login',
    'https://www.googleapis.com/auth/userinfo.email',
    'https://www.googleapis.com/auth/userinfo.profile'
]
```

To ease the change, the old API and scopes is still supported by the application, the new values are the default option but if having troubles supporting them you can default to the old values by defining this setting:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_USE_DEPRECATED_API = True
SOCIAL_AUTH_GOOGLE_PLUS_USE_DEPRECATED_API = True
```

Instagram

Instagram uses OAuth v2 for Authentication.

- Register a new application at the Instagram API, and
- Add instagram backend to `AUTHENTICATION_SETTINGS`:

```
AUTHENTICATION_SETTINGS = (
  ...
  'social_core.backends.instagram.InstagramOAuth2',
  ...
)
```

- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_INSTAGRAM_KEY = ''
SOCIAL_AUTH_INSTAGRAM_SECRET = ''
```

- extra scopes can be defined by using:

```
SOCIAL_AUTH_INSTAGRAM_AUTH_EXTRA_ARGUMENTS = {'scope': 'likes comments␣
↪relationships'}
```

Itembase

Itembase uses OAuth2 for authentication.

- Register a new application for the Itembase API, and
- Add itembase live backend and/or sandbox backend to `AUTHENTICATION_BACKENDS`:

```
AUTHENTICATION_BACKENDS = (
  ...
  'social_core.backends.itembase.ItembaseOAuth2',
  'social_core.backends.itembase.ItembaseOAuth2Sandbox',
  ...
)
```

- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_ITEMBASE_KEY = ''
SOCIAL_AUTH_ITEMBASE_SECRET = ''

SOCIAL_AUTH_ITEMBASE_SANDBOX_KEY = ''
SOCIAL_AUTH_ITEMBASE_SANDBOX_SECRET = ''
```

- extra scopes can be defined by using:

```
SOCIAL_AUTH_ITEMBASE_SCOPE = ['connection.transaction',
                              'connection.product',
                              'connection.profile',
                              'connection.buyer']
SOCIAL_AUTH_ITEMBASE_SANDBOX_SCOPE = SOCIAL_AUTH_ITEMBASE_SCOPE
```

To use data from the extra scopes, you need to do an extra activation step that is not in the usual OAuth flow. For that you can extend your pipeline and add a function that sends the user to an activation URL that Itembase provides. The method to retrieve the activation data is included in the backend:

```
@partial
def activation(strategy, backend, response, *args, **kwargs):
    if backend.name.startswith("itembase"):

        if strategy.session_pop('itembase_activation_in_progress'):
            strategy.session_set('itembase_activated', True)

        if not strategy.session_get('itembase_activated'):
            activation_data = backend.activation_data(response)
            strategy.session_set('itembase_activation_in_progress', True)
            return HttpResponseRedirect(activation_data['activation_url'])
```

Jawbone

Jawbone uses OAuth2. In order to enable the backend follow:

- Register an application at Jawbone Developer Portal, set the `OAuth redirect URIs` to `http://<your hostname>/complete/jawbone/`
- Fill in the **Client Id** and **Client Secret** values in your settings:

```
SOCIAL_AUTH_JAWBONE_KEY = ''
SOCIAL_AUTH_JAWBONE_SECRET = ''
```

- Specify scopes with:

```
SOCIAL_AUTH_JAWBONE_SCOPE = [...]
```

  Available scopes are listed in the Jawbone Authentication Reference, "socpes" section.

Just Giving

OAuth2

Follow the steps at Just Giving API Docs to register your application and get the needed keys.

- Add the Just Giving OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.justgiving.JustGivingOAuth2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_JUSTGIVING_KEY = ''
SOCIAL_AUTH_JUSTGIVING_SECRET = ''
```

Kakao

Kakao uses OAuth v2 for Authentication.

- Register a new applicationat the Kakao API, and

- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_KAKAO_KEY = ''
SOCIAL_AUTH_KAKAO_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_KAKAO_SCOPE = [...]
```

Khan Academy

Khan Academy uses a variant of OAuth1 authentication flow. Check the API details at Khan Academy API Authentication.

Follow this steps in order to use the backend:

- Register a new application at Khan Academy API Apps,
- Fill **Consumer Key** and **Consumer Secret** values:

```
SOCIAL_AUTH_KHANACADEMY_OAUTH1_KEY = ''
SOCIAL_AUTH_KHANACADEMY_OAUTH1_SECRET = ''
```

- Add the backend to `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.khanacademy.KhanAcademyOAuth1',
    ...
)
```

Last.fm

Last.fm uses a similar authentication process than OAuth2 but it's not. In order to enable the support for it just:

- Register an application at Get an API Account, set the Last.fm callback to something sensible like http://your.site/complete/lastfm
- Fill in the **API Key** and **API Secret** values in your settings:

```
SOCIAL_AUTH_LASTFM_KEY = ''
SOCIAL_AUTH_LASTFM_SECRET = ''
```

- Enable the backend in `AUTHENTICATION_BACKENDS` setting.

Launchpad

Ubuntu Launchpad OpenId doesn't require major settings beside being defined on `AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.launchpad.LaunchpadOpenId',
    ...
)
```

Line.me

Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_LINE_KEY = '...'
SOCIAL_AUTH_LINE_SECRET = '...'
```

LinkedIn

LinkedIn supports OAuth1 and OAuth2. Migration between each type is fair simple since the same Key / Secret pair is used for both authentication types.

LinkedIn OAuth setup is similar to any other OAuth service. The auth flow is explained on LinkedIn Developers docs. First you will need to register an app att LinkedIn Developer Network.

OAuth1

- Fill the application key and secret in your settings:

  ```
  SOCIAL_AUTH_LINKEDIN_KEY = ''
  SOCIAL_AUTH_LINKEDIN_SECRET = ''
  ```

- Application scopes can be specified by:

  ```
  SOCIAL_AUTH_LINKEDIN_SCOPE = [...]
  ```

  Check the available options at LinkedIn Scopes. If you want to request a user's email address, you'll need specify that your application needs access to the email address use the `r_emailaddress` scope.

- To request extra fields using LinkedIn fields selectors just define this setting:

  ```
  SOCIAL_AUTH_LINKEDIN_FIELD_SELECTORS = [...]
  ```

  with the needed fields selectors, also define `SOCIAL_AUTH_LINKEDIN_EXTRA_DATA` properly as described in OAuth, that way the values will be stored in `UserSocialAuth.extra_data` field. By default `id`, `first-name` and `last-name` are requested and stored.

For example, to request a user's email, headline, and industry from the Linkedin API and store the information in `UserSocialAuth.extra_data`, you would add these settings:

```
# Add email to requested authorizations.
SOCIAL_AUTH_LINKEDIN_SCOPE = ['r_basicprofile', 'r_emailaddress', ...]
# Add the fields so they will be requested from linkedin.
SOCIAL_AUTH_LINKEDIN_FIELD_SELECTORS = ['email-address', 'headline', 'industry']
# Arrange to add the fields to UserSocialAuth.extra_data
SOCIAL_AUTH_LINKEDIN_EXTRA_DATA = [('id', 'id'),
                                   ('firstName', 'first_name'),
                                   ('lastName', 'last_name'),
                                   ('emailAddress', 'email_address'),
                                   ('headline', 'headline'),
                                   ('industry', 'industry')]
```

OAuth2

OAuth2 works exacly the same than OAuth1, but the settings must be named as:

```
SOCIAL_AUTH_LINKEDIN_OAUTH2_*
```

Looks like LinkedIn is forcing the definition of the callback URL in the application when OAuth2 is used. Be sure to set the proper values, otherwise a `(400) Client Error: Bad Request` might be returned by their service.

LiveJournal

LiveJournal provides OpenId, it doesn't require any major settings in order to work, beside being defined on `AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.aol.AOLOpenId',
    ...
)
```

LiveJournal OpenId is provided by URLs in the form `http://<username>.livejournal.com`, this application retrieves the `username` from the data in the current request by checking a parameter named `openid_lj_user` which can be sent by `POST` or `GET`.

MSN Live Connect

Live uses OAuth2 for its connect workflow, notice that it isn't OAuth WRAP.

- Register a new application at Live Connect Developer Center, set your site domain as redirect domain,
- Fill `Client Id` and `Client Secret` values in the settings:

  ```
  SOCIAL_AUTH_LIVE_KEY = ''
  SOCIAL_AUTH_LIVE_SECRET = ''
  ```

- Also it's possible to define extra permissions with:

  ```
  SOCIAL_AUTH_LIVE_SCOPE = [...]
  ```

  Defaults are `wl.basic` and `wl.emails`. Latter one is necessary to retrieve user email.

- Ensure to have a valid `Redirect URL` (`http://your-domain/complete/live`) defined in the application if `Enhanced security redirection` is enabled.

LoginRadius

LoginRadius uses OAuth2 for Authentication with other providers with an HTML widget used to trigger the auth process.

- Register a new application at the LoginRadius Website, and
- Fill `Client Id` and `Client Secret` values in the settings:

  ```
  SOCIAL_AUTH_LOGINRADIUS_KEY = ''
  SOCIAL_AUTH_LOGINRADIUS_SECRET = ''
  ```

- Since the auth process is triggered by LoginRadius JS script, you need to sever such content to the user, all you need to do that is a template with the following content:

```
<div id="interfacecontainerdiv" class="interfacecontainerdiv"></div>
<script src="https://hub.loginradius.com/include/js/LoginRadius.js"></script>
<script type="text/javascript">
    var options = {};
    options.login = true;
    LoginRadius_SocialLogin.util.ready(function () {
        $ui = LoginRadius_SocialLogin.lr_login_settings;
        $ui.interfacesize = "";
        $ui.apikey = "{{ LOGINRADIUS_KEY }}";
        $ui.callback = "{{ LOGINRADIUS_REDIRECT_URL }}";
        $ui.lrinterfacecontainer = "interfacecontainerdiv";
        LoginRadius_SocialLogin.init(options);
    });
</script>
```

  Put that content in a template named `loginradius.html` (accessible to your framework), or define a name with `SOCIAL_AUTH_LOGINRADIUS_TEMPLATE` setting, like:

```
SOCIAL_AUTH_LOGINRADIUS_LOCAL_HTML = 'loginradius.html'
```

  The template context will have the current backend instance under the `backend` name, also the application key (`LOGINRADIUS_KEY`) and the redirect URL (`LOGINRADIUS_REDIRECT_URL`).

- Further documentation can be found at LoginRadius API Documentation and LoginRadius Datapoints

Lyft

Lyft implements OAuth2 as its authorization service. To setup a Lyft backend:

1. Register a new application via the Lyft Developer Portal.
2. Add the Lyft OAuth2 backend as an option in your settings:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.lyft.LyftOAuth2',
    ...
)
```

3. Use the `Client Id` and `Client Secret` from the Developer Portal into your settings:

```
SOCIAL_AUTH_LYFT_KEY = ''
SOCIAL_AUTH_LYFT_SECRET = ''
```

4. Specify the scope that your app should have access to:

```
SOCIAL_AUTH_LYFT_SCOPE = ['public', 'profile', 'rides.read', 'rides.request']
```

To learn more about the API and the calls that are available, read the Lyft API Documentation.

MailChimp

MailChimp uses OAuth v2 for Authentication, check the official docs.

- Create an app by filling out the form here: Add App

- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_MAILCHIMP_KEY = '<App UID>'
SOCIAL_AUTH_MAILCHIMP_SECRET = '<App secret>'
```

- Add the backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.mailchimp.MailChimpOAuth2',
    ...
)
```

- Then you can start using `{% url social_core:begin 'mailchimp' %}` in your templates

Mail.ru OAuth

Mail.ru uses OAuth2 workflow, to use it fill in settings:

```
SOCIAL_AUTH_MAILRU_OAUTH2_KEY = ''
SOCIAL_AUTH_MAILRU_OAUTH2_SECRET = ''
```

MapMyFitness

MapMyFitness uses OAuth v2 for authentication.

- Register a new application at the MapMyFitness API, and
- fill `key` and `secret` values in the settings:

```
SOCIAL_AUTH_MAPMYFITNESS_KEY = ''
SOCIAL_AUTH_MAPMYFITNESS_SECRET = ''
```

Meetup

Meetup.com uses OAuth2 for its auth mechanism.

- Register a new OAuth Consumer at Meetup Consumer Registration, set your consumer name, redirect uri.
- Fill `key` and `secret` values in the settings:

```
SOCIAL_AUTH_MEETUP_KEY = ''
SOCIAL_AUTH_MEETUP_SECRET = ''
```

Mendeley

Mendeley supports OAuth1 and OAuth2, they are in the process of deprecating OAuth1 API (which should be fully deprecated on April 2014, check their announcement).

OAuth1

In order to support OAuth1 (not recomended, use OAuth2 instead):

- Register a new application at Mendeley Application Registration
- Fill **Consumer Key** and **Consumer Secret** values:

```
SOCIAL_AUTH_MENDELEY_KEY = ''
SOCIAL_AUTH_MENDELEY_SECRET = ''
```

OAuth2

In order to support OAuth2:

- Register a new application at Mendeley Application Registration, or migrate your OAuth1 application, check their migration steps here.
- Fill **Application ID** and **Application Secret** values:

```
SOCIAL_AUTH_MENDELEY_OAUTH2_KEY = ''
SOCIAL_AUTH_MENDELEY_OAUTH2_SECRET = ''
```

MineID

MineID works similar to Facebook (OAuth).

- Register a new application at MineID.org, set the callback URL to `http://example.com/complete/mineid/` replacing `example.com` with your domain.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_MINEID_KEY = ''
SOCIAL_AUTH_MINEID_SECRET = ''
```

Self-hosted MineID

Since MineID is an Open Source software and can be self-hosted, you can change settings to point to your instance:

```
SOCIAL_AUTH_MINEID_HOST = 'www.your-mineid-instance.com'
SOCIAL_AUTH_MINEID_SCHEME = 'https'  # or 'http'
```

Mixcloud OAuth2

The Mixcloud API offers support for authorization. To this backend support:

- Register a new application at Mixcloud Developers
- Add Mixcloud backend to `AUTHENTICATION_BACKENDS` in settings:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.mixcloud.MixcloudOAuth2',
)
```

- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_MIXCLOUD_KEY = ''
SOCIAL_AUTH_MIXCLOUD_SECRET = ''
```

- Similar to the other OAuth backends you can define:

```
SOCIAL_AUTH_MIXCLOUD_EXTRA_DATA = [('username', 'username'),
                                   ('name', 'name'),
                                   ('pictures', 'pictures'),
                                   ('url', 'url')]
```

  as a list of tuples `(response name, alias)` to store user profile data on the `UserSocialAuth.extra_data`.

Moves

Moves provides an OAuth2 authentication flow. In order to enable it:

- Register an application at Manage Your Apps, remember to fill the `Redirect URI` once the application was created.
- Fill **Client ID** and **Client secret** in the settings:

```
SOCIAL_AUTH_MOVES_KEY = ''
SOCIAL_AUTH_MOVES_SECRET = ''
```

- Define the mandatory scope for your application:

```
SOCIAL_AUTH_MOVES_SCOPE = ['activity', 'location']
```

  The scope parameter is required by Moves but the backend doesn't set a default one to minimize the application permissions request, so it's mandatory for the developer to define this setting.

- Add the backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.moves.MovesOAuth2',
    ...
)
```

NationBuilder

NaszaKlasa supports OAuth2 as their authentication mechanism. Follow these steps in order to use it:

- Register a new application at your NK Developers (define the *Callback URL* to `http://example.com/complete/nk/` where `example.com` is your domain).
- Fill the `Client ID` and `Client Secret` values from the newly created application:

```
SOCIAL_AUTH_NK_KEY = ''
SOCIAL_AUTH_NK_SECRET = ''
```

- Enable the backend in `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.nk.NKOAuth2',
    ...
)
```

NationBuilder

NationBuilder supports OAuth2 as their authentication mechanism. Follow these steps in order to use it:

- Register a new application at your Nation Admin panel (define the *Callback URL* to `http://example.com/complete/nationbuilder/` where `example.com` is your domain).
- Fill the `Client ID` and `Client Secret` values from the newly created application:

```
SOCIAL_AUTH_NATIONBUILDER_KEY = ''
SOCIAL_AUTH_NATIONBUILDER_SECRET = ''
```

- Also define your NationBuilder slug:

```
SOCIAL_AUTH_NATIONBUILDER_SLUG = 'your-nationbuilder-slug'
```

- Enable the backend in `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.nationbuilder.NationBuilderOAuth2'
    ...
)
```

Naver

Naver uses OAuth v2 for Authentication.

- Register a new application at the Naver API, and
- add naver oauth backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.naver.NaverOAuth2',
    ...
)
```

- fill `Client ID` and `Client Secret` from developer.naver.com values in the settings:

```
SOCIAL_AUTH_NAVER_KEY = ''
SOCIAL_AUTH_NAVER_SECRET = ''
```

- you can get EXTRA_DATA:

```
SOCIAL_AUTH_NAVER_EXTRA_DATA = ['nickname', 'gender', 'age',
                                'birthday', 'profile_image']
```

NGP VAN ActionID

NGP VAN's ActionID service provides an OpenID 1.1 endpoint, which provides first name, last name, email address, and phone number.

ActionID doesn't require major settings beside being defined on `AUTHENTICATION_BACKENDS`

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.ngpvan.ActionIDOpenID',
    ...
)
```

If you want to be able to access the "phone" attribute offered by NGP VAN within `extra_data` you can add the following to your settings:

```
SOCIAL_AUTH_ACTIONID_OPENID_AX_EXTRA_DATA = [
    ('http://openid.net/schema/contact/phone/business', 'phone')
]
```

NGP VAN offers the ability to have your domain whitelisted, which will disable the "{domain} is requesting a link to your ActionID" warning when your app attempts to login using an ActionID account. Contact NGP VAN Developer Support for more information

Odnoklassniki.ru

There are two options with Odnoklassniki: either you use OAuth2 workflow to authenticate odnoklassniki users at external site, or you authenticate users within your IFrame application.

OAuth2

If you use OAuth2 workflow, you need to:

- register a new application with OAuth registration form
- fill out some settings:

```
SOCIAL_AUTH_ODNOKLASSNIKI_OAUTH2_KEY = ''
SOCIAL_AUTH_ODNOKLASSNIKI_OAUTH2_SECRET = ''
SOCIAL_AUTH_ODNOKLASSNIKI_OAUTH2_PUBLIC_NAME = ''
```

- add `'social_core.backends.odnoklassniki.OdnoklassnikiOAuth2'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.

IFrame applications

If you want to authenticate users in your IFrame application,

- read Rules for application developers
- fill out Developers registration form
- get your personal sandbox
- fill out some settings:

```
SOCIAL_AUTH_ODNOKLASSNIKI_APP_KEY = ''
SOCIAL_AUTH_ODNOKLASSNIKI_APP_SECRET = ''
SOCIAL_AUTH_ODNOKLASSNIKI_APP_PUBLIC_NAME = ''
```

- add `'social_core.backends.odnoklassniki.OdnoklassnikiApp'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`
- sign a public offer and do some bureaucracy

You may also use:

```
SOCIAL_AUTH_ODNOKLASSNIKI_APP_EXTRA_USER_DATA_LIST
```

Defaults to empty tuple, for the list of available fields see Documentation on user.getInfo

OpenStreetMap

OpenStreetMap supports OAuth 1.0 and 1.0a but 1.0a should be used for the new applications, as 1.0 is for support of legacy clients only.

Access tokens currently do not expire automatically.

More documentation at OpenStreetMap Wiki:

- Login to your account
- Register your application as OAuth consumer on your OpenStreetMap user settings page, and
- Set `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_OPENSTREETMAP_KEY = ''
SOCIAL_AUTH_OPENSTREETMAP_SECRET = ''
```

Orbi

Orbi OAuth v2 for Authentication.

- Register a new applicationat the Orbi API, and
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_ORBI_KEY = ''
SOCIAL_AUTH_ORBI_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_KAKAO_SCOPE = ['all']
```

Mozilla Persona

Support for Mozilla Persona is possible by posting the `assertion` code to `/complete/persona/` URL.

The setup doesn't need any setting, just the usual Mozilla Persona javascript include in your document and the needed mechanism to trigger the POST to python-social-auth:

```
<!-- Include BrowserID JavaScript -->
<script src="https://login.persona.org/include.js" type="text/javascript"></script>

<!-- Define a form to send the POST data -->
<form method="post" action="/complete/persona/">
    <input type="hidden" name="assertion" value="" />
    <a rel="nofollow" id="persona" href="#">Mozilla Persona</a>
</form>

<!-- Setup click handler that retieves Persona assertion code and sends POST data -->
<script type="text/javascript">
    $(function () {
        $('#persona').click(function (e) {
            e.preventDefault();
            var self = $(this);

            navigator.id.get(function (assertion) {
                if (assertion) {
                    self.parent('form')
                            .find('input[type=hidden]')
                                .attr('value', assertion)
                                .end()
                            .submit();
                } else {
                    alert('Some error occurred');
                }
            });
        });
    });
</script>
```

Pinterest

Pinterest implemented OAuth2 protocol for their authentication mechanism. To enable `python-social-auth` support follow this steps:

1. Go to Pinterest developers zone and create an application.
2. Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_PINTEREST_KEY = '...'
SOCIAL_AUTH_PINTEREST_SECRET = '...'
SOCIAL_AUTH_PINTEREST_SCOPE = [
    'read_public',
    'write_public',
    'read_relationships',
    'write_relationships'
]
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.pinterest.PinterestOAuth2',
    ...
)
```

PixelPin

PixelPin only supports OAuth2.

PixelPin OAuth2

Developer documentation for PixelPin can be found at http://developer.pixelpin.co.uk/. To setup OAuth2 do the following:

- Register a new developer account at PixelPin Developers.

  You require a PixelPin account to create developer accounts. Sign up at PixelPin Account Page For the value of redirect uri, use whatever path you need to return to on your web application. The example code provided with the plugin uses `http://<yoursite>/complete/pixelpin-oauth2/`.

  Once verified by email, record the values of client id and secret for the next step.

- Fill **Consumer Key** and **Consumer Secret** values in your settings.py file:

```
SOCIAL_AUTH_PIXELPIN_OAUTH2_KEY = ''
SOCIAL_AUTH_PIXELPIN_OAUTH2_SECRET = ''
```

- Add `'social_core.backends.pixelpin.PixelPinOAuth2'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.

Pocket

Pocket uses a weird variant of OAuth v2 that only defines a consumer key.

- Register a new application at the Pocket API, and
- fill `consumer key` value in the settings:

```
SOCIAL_AUTH_POCKET_KEY = ''
```

Podio

Podio offers OAuth2 as their auth mechanism. In order to enable it, follow:

- Register a new application at Podio API Keys
- Fill **Client Id** and **Client Secret** values:

```
SOCIAL_AUTH_PODIO_KEY = ''
SOCIAL_AUTH_PODIO_SECRET = ''
```

Qiita

Qiita

- Register a new application at Qiita, set the callback URL to `http://example.com/complete/qiita/` replacing `example.com` with your domain.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_QIITA_KEY = ''
SOCIAL_AUTH_QIITA_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_QIITA_SCOPE = [...]
```

See auth scopes at Qiita Scopes docs.

QQ

QQ implemented OAuth2 protocol for their authentication mechanism. To enable `python-social-auth` support follow this steps:

1. Go to QQ and create an application.
2. Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_QQ_KEY = '...'
SOCIAL_AUTH_QQ_SECRET = '...'
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.qq.QQOauth2',
    ...
)
```

The values for `nickname`, `figureurl_qq_1` and `gender` will be stored in the `extra_data` field. The `nickname` will be used as the account username. `figureurl_qq_1` can be used as the profile image.

Sometimes nickname will duplicate with another `qq` account, to avoid this issue it's possible to use `openid` as `username` by define this setting:

```
SOCIAL_AUTH_QQ_USE_OPENID_AS_USERNAME = True
```

Quizlet

Quizlet uses OAuth v2 for Authentication.

- Register a new application at the Quizlet API, and
- Add the Quizlet backend to `AUTHENTICATION_SETTINGS`:

```
AUTHENTICATION_SETTINGS = (
  ...
  'social_core.backends.quizlet.QuizletOAuth2',
  ...
)
```

- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_QUIZLET_KEY = ''
SOCIAL_AUTH_QUIZLET_SECRET = ''
SOCIAL_AUTH_QUIZLET_SCOPE = ['read', 'write_set']  # 'write_group' is also␣
↪available
```

Rdio

Rdio provides OAuth 1 and 2 support for their authentication process.

OAuth 1.0a

To setup Rdio OAuth 1.0a, add the following to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.rdio.RdioOAuth1',
    ...
)

SOCIAL_AUTH_RDIO_OAUTH1_KEY = ''
SOCIAL_AUTH_RDIO_OAUTH1_SECRET = ''
```

OAuth 2.0

To setup Rdio OAuth 2.0, add the following to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.rdio.RdioOAuth2',
    ...
)

SOCIAL_AUTH_RDIO_OAUTH2_KEY = os.environ['RDIO_OAUTH2_KEY']
SOCIAL_AUTH_RDIO_OAUTH2_SECRET = os.environ['RDIO_OAUTH2_SECRET']
SOCIAL_AUTH_RDIO_OAUTH2_SCOPE = []
```

Extra Fields

The following extra fields are automatically requested:

- rdio_id
- rdio_icon_url
- rdio_profile_url
- rdio_username
- rdio_stream_region

Readability

Readability works similarly to Twitter, in that you'll need a `Consumer Key` and `Consumer Secret`. These can be obtained in the `Connections` section of your `Account` page.

- Fill the **Consumer Key** and **Consumer Secret** values in your settings:

```
SOCIAL_AUTH_READABILITY_KEY = ''
SOCIAL_AUTH_READABILITY_SECRET = ''
```

That's it! By default you'll get back:

```
username
first_name
last_name
```

with EXTRA_DATA, you can get:

```
date_joined
kindle_email_address
avatar_url
email_into_address
```

Reddit

Reddit implements OAuth2 authentication workflow. To enable it, just follow:

- Register an application at Reddit Preferences Apps
- Fill the **Consumer Key** and **Consumer Secret** values in your settings:

```
SOCIAL_AUTH_REDDIT_KEY = ''
SOCIAL_AUTH_REDDIT_SECRET = ''
```

- By default the token is not permanent, it will last an hour. To get a refresh token just define:

```
SOCIAL_AUTH_REDDIT_AUTH_EXTRA_ARGUMENTS = {'duration': 'permanent'}
```

  This will store the `refresh_token` in `UserSocialAuth.extra_data` attribute, to refresh the access token just do:

```
from social_django.utils import load_strategy

strategy = load_strategy(backend='reddit')
user = User.objects.get(pk=foo)
social = user.social_auth.filter(provider='reddit')[0]
social.refresh_token(strategy=strategy,
                     redirect_uri='http://localhost:8000/complete/reddit/')
```

  Reddit requires `redirect_uri` when refreshing the token and it must be the same value used during the auth process.

RunKeeper

RunKeeper uses OAuth v2 for authentication.

- Register a new application at the RunKeeper API, and
- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_RUNKEEPER_KEY = ''
SOCIAL_AUTH_RUNKEEPER_SECRET = ''
```

Salesforce

Salesforce uses OAuth v2 for Authentication, check the official docs.

- Create an app following the steps in the Defining Connected Apps docs.
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_SALESFORCE_OAUTH2_KEY = '<App UID>'
SOCIAL_AUTH_SALESFORCE_OAUTH2_SECRET = '<App secret>'
```

- Add the backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.salesforce.SalesforceOAuth2',
    ...
)
```

- Then you can start using `{% url social:begin 'salesforce-oauth2' %}` in your templates

If using the sandbox mode:

- Fill these settings instead:

```
SOCIAL_AUTH_SALESFORCE_OAUTH2_SANDBOX_KEY = '<App UID>'
SOCIAL_AUTH_SALESFORCE_OAUTH2_SANDBOX_SECRET = '<App secret>'
```

- And this backend:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.salesforce.SalesforceOAuth2Sandbox',
    ...
)
```

- Then you can start using `{% url social:begin 'salesforce-oauth2-sandbox' %}` in your templates

Shimmering Verify

Shimmering implemented OAuth2 protocol for their authentication mechanism. To enable `python-social_core-auth` support follow this steps:

1. Go to Shimmering Developer Console and create an application.
2. Fill App Id and Secret in your project settings:

```
SOCIAL_AUTH_SHIMMERING_KEY = '...'
SOCIAL_AUTH_SHIMMERING_SECRET = '...'
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.shimmering.ShimmeringOAuth2',
    ...
)
```

Shopify

Shopify uses OAuth 2 for authentication.

To use this backend, you must install the package `shopify` from the Github project. Currently supports v2+

- Register a new application at Shopify Partners, and
- Set the Auth Type to OAuth2 in the application settings
- Set the Application URL to http://{[}your domain]/login/shopify/
- fill `API Key` and `Shared Secret` values in your django settings:

```
SOCIAL_AUTH_SHOPIFY_KEY    = ''
SOCIAL_AUTH_SHOPIFY_SECRET = ''
```

- fill the scope permissions that you require into the settings Shopify API:

```
SOCIAL_AUTH_SHOPIFY_SCOPE = ['write_script_tags',
                             'read_orders',
                             'write_customers',
                             'read_products']
```

Sketchfab

Sketchfab uses OAuth 2 for authentication.

To use:

- Follow the steps at Sketchfab Oauth, and ask for an `Authorization code` grant type.
- Fill the `Client id/key` and `Client Secret` values you received in your django settings:

```
SOCIAL_AUTH_SKETCHFAB_KEY    = ''
SOCIAL_AUTH_SKETCHFAB_SECRET = ''
```

Skyrock

OAuth based Skyrock Connect.

Skyrock offers per application keys named `Consumer Key` and `Consumer Secret`. To enable Skyrock these two keys are needed. Further documentation at Skyrock developer resources:

- Register a new application at Skyrock App Creation,
- Your callback domain should match your application URL in your application configuration.
- Fill **Consumer Key** and **Consumer Secret** values:

```
SOCIAL_AUTH_SKYROCK_KEY = ''
SOCIAL_AUTH_SKYROCK_SECRET = ''
```

Slack

Slack

- Register a new application at Slack, set the callback URL to `http://example.com/complete/slack/` replacing `example.com` with your domain.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_SLACK_KEY = ''
SOCIAL_AUTH_SLACK_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_SLACK_SCOPE = [...]
```

See auth scopes at Slack OAuth docs.

SoundCloud

SoundCloud uses OAuth2 for its auth mechanism.

- Register a new application at SoundCloud App Registration, set your application name, website and redirect URI.
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_SOUNDCLOUD_KEY = ''
SOCIAL_AUTH_SOUNDCLOUD_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_SOUNDCLOUD_SCOPE = [...]
```

Possible scope values are * or *non-expiring* according to their /connect documentation.

Check the rest of their doc at SoundCloud Developer Documentation.

Spotify

Spotify supports OAuth 2.

- Register a new application at Spotify Web API, and follow the instructions below.

OAuth2

Add the Spotify OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.spotify.SpotifyOAuth2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_SPOTIFY_KEY = ''
SOCIAL_AUTH_SPOTIFY_SECRET = ''
```

SUSE

This section describes how to setup the different services provided by SUSE and openSUSE.

openSUSE OpenId

openSUSE OpenId works straightforward, not settings are needed. Domains or emails whitelists can be applied too, check the whitelists settings for details.

Stackoverflow

Stackoverflow uses OAuth 2.0

- "Register For An App Key" at the Stack Exchange API site. Set your OAuth domain and application website settings.
- Add the `Client Id`, `Client Secret` and `API Key` values in settings:

```
SOCIAL_AUTH_STACKOVERFLOW_KEY = ''
SOCIAL_AUTH_STACKOVERFLOW_SECRET = ''
SOCIAL_AUTH_STACKOVERFLOW_API_KEY = ''
```

- You can ask for extra permissions with:

```
SOCIAL_AUTH_STACKOVERFLOW_SCOPE = [...]
```

Steam OpenId

Steam OpenId works quite straightforward, but to retrieve some user data (known as `player` on Steam API) a Steam API Key is needed.

Configurable settings:

- Supply a Steam API Key from Steam Dev:

```
SOCIAL_AUTH_STEAM_API_KEY = key
```

- To save `player` data provided by Steam into `extra_data`:

```
SOCIAL_AUTH_STEAM_EXTRA_DATA = ['player']
```

StockTwits

StockTwits uses OAuth 2 for authentication.

- Register a new application at https://stocktwits.com/developers/apps
- Set the Website URL to http://{[}your domain]/
- fill `Consumer Key` and `Consumer Secret` values in your django settings:

```
SOCIAL_AUTH_STOCKTWITS_KEY    = ''
SOCIAL_AUTH_STOCKTWITS_SECRET = ''
```

Strava

Strava uses OAuth v2 for Authentication.

- Register a new application at the Strava API, and

- fill `Client ID` and `Client Secret` from strava.com values in the settings:

```
SOCIAL_AUTH_STRAVA_KEY = ''
SOCIAL_AUTH_STRAVA_SECRET = ''
```

- extra scopes can be defined by using:

```
SOCIAL_AUTH_STRAVA_SCOPE = ['view_private']
```

Stripe

Stripe uses OAuth2 for its authorization service. To setup Stripe backend:

- Register a new application at Stripe App Creation, and
- Grab the `client_id` value in `Applications` tab and fill the `App Id` setting:

```
SOCIAL_AUTH_STRIPE_KEY = 'ca_...'
```

- Grab the `Test Secret Key` in the `API Keys` tab and fille the `App Secret` setting:

```
SOCIAL_AUTH_STRIPE_SECRET = '...'
```

- Define `SOCIAL_AUTH_STRIPE_SCOPE` with the desired scope (options are `read_only` and `read_write`):

```
SOCIAL_AUTH_STRIPE_SCOPE = ['read_only']
```

- Add the needed backend to `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.stripe.StripeOAuth2',
    ...
)
```

More info on Stripe OAuth2 at Integrating OAuth.

Taobao OAuth

Taobao OAuth 2.0 workflow.

- Register a new application at Open Open Taobao.
- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_TAOBAO_KEY = ''
SOCIAL_AUTH_TAOBAO_SECRET = ''
```

By default `token` is stored in `extra_data` field.

ThisIsMyJam

ThisIsMyJam uses OAuth1 for its auth mechanism.

- Register a new application at ThisIsMyJam App Registration, set your application name, website and redirect URI.

- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_THISISMYJAM_KEY = ''
SOCIAL_AUTH_THISISMYJAM_SECRET = ''
```

Check the rest of their doc at ThisIsMyJam API Docs.

Trello

Trello provides OAuth1 support for their authentication process.

In order to enable it, follow:

- Generate an Application Key pair at Trello Developers API Keys
- Fill **Consumer Key** and **Consumer Secret** settings:

```
SOCIAL_AUTH_TRELLO_KEY = '...'
SOCIAL_AUTH_TRELLO_SECRET = '...'
```

There are also two optional settings:

- your app name, otherwise the authorization page will say "Let An unknown application use your account?":

```
SOCIAL_AUTH_TRELLO_APP_NAME = 'My App'
```

- the expiration period, social auth defaults to 'never', but you can change it:

```
SOCIAL_AUTH_TRELLO_EXPIRATION = '30days'
```

TripIt

TripIt offers per application keys named `API Key` and `API Secret`. To enable TripIt these two keys are needed. Further documentation at TripIt Developer Center:

- Register a new application at TripIt App Registration,
- fill **API Key** and **API Secret** values:

```
SOCIAL_AUTH_TRIPIT_KEY = ''
SOCIAL_AUTH_TRIPIT_SECRET = ''
```

Tumblr

Tumblr uses OAuth 1.0a for authentication.

- Register a new application at http://www.tumblr.com/oauth/apps
- Set the `Default callback URL` to http://{[}your domain]/
- fill `OAuth Consumer Key` and `Secret Key` values in your Django settings:

```
SOCIAL_AUTH_TUMBLR_KEY = ''
SOCIAL_AUTH_TUMBLR_SECRET = ''
```

Twilio

- Register a new application at Twilio Connect Api
- Fill `SOCIAL_AUTH_TWILIO_KEY` and `SOCIAL_AUTH_TWILIO_SECRET` values in the settings:

```
SOCIAL_AUTH_TWILIO_KEY = ''
SOCIAL_AUTH_TWILIO_SECRET = ''
```

- Add desired authentication backends to Django's `SOCIAL_AUTH_AUTHENTICATION_BACKENDS` setting:

```
'social_core.backends.twilio.TwilioAuth',
```

- Usage example:

```
<a href="/login/twilio">Enter using Twilio</a>
```

Twitch

Twitch works similar to Facebook (OAuth).

- Register a new application in the connections tab of your Twitch settings page, set the callback URL to `http://example.com/complete/twitch/` replacing `example.com` with your domain.
- Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_TWITCH_KEY = ''
SOCIAL_AUTH_TWITCH_SECRET = ''
```

- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_TWITCH_SCOPE = [...]
```

Twitter

Twitter offers per application keys named `Consumer Key` and `Consumer Secret`. To enable Twitter these two keys are needed. Further documentation at Twitter development resources:

- Register a new application at Twitter App Creation,
- Check the **Allow this application to be used to Sign in with Twitter** checkbox. If you don't check this box, Twitter will force your user to login every time.
- Fill **Consumer Key** and **Consumer Secret** values:

```
SOCIAL_AUTH_TWITTER_KEY = ''
SOCIAL_AUTH_TWITTER_SECRET = ''
```

- You need to specify an URL callback or the application will be marked as Client type instead of the Browser. Almost any dummy value will work if you plan some test.
- You can request user's Email address (consult Twitter verify credentials), the parameter is sent automatically, but the applicaton needs to be whitelisted in order to get a valid value.

Twitter usually fails with a 401 error when trying to call the request-token URL, this is usually caused by server datetime errors (check miscellaneous section). Installing `ntp` and syncing the server date with some pool does the trick.

Uber

Uber uses OAuth v2 for Authentication.

- Register a new application at the Uber API, and follow the instructions below

OAuth2

1. Add the Uber OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.uber.UberOAuth2',
    ...
)
```

2. Fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_UBER_KEY = ''
SOCIAL_AUTH_UBER_SECRET = ''
```

3. Scope should be defined by using:

```
SOCIAL_AUTH_UBER_SCOPE = ['profile', 'request']
```

Untappd

Untappd uses OAuth v2 for Authentication, check the official docs.

- Create an app by filling out the form here: Add App
- Apps are approved on a one-by-one basis, so you'll need to wait a few days to get your client ID and secret.
- Fill `Client ID` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_UNTAPPD_KEY = '<App UID>'
SOCIAL_AUTH_UNTAPPD_SECRET = '<App secret>'
```

- Add the backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.untappd.UntappdOAuth2',
    ...
)
```

- Then you can start using `{% url social:begin 'untappd' %}` in your templates

Upwork

Upwork supports only OAuth 1.

- Register a new application at Upwork Developers.

OAuth1

Add the Upwork OAuth backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.upwork.UpworkOAuth',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_UPWORK_KEY = ''
SOCIAL_AUTH_UPWORK_SECRET = ''
```

Note: For more information please go to Upwork API Reference.

Vend

Vend supports OAuth 2.

- Register a new application at Vend Developers Portal
- Add the Vend OAuth2 backend to your settings page:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.vend.VendOAuth2',
    ...
)
```

- Fill `App Key` and `App Secret` values in the settings:

```
SOCIAL_AUTH_VEND_OAUTH2_KEY = ''
SOCIAL_AUTH_VEND_OAUTH2_SECRET = ''
```

More details on their docs.

Vimeo

Vimeo uses OAuth1 to grant access to their API. In order to get the backend running follow:

- Register an application at Vimeo Developer Portal filling the required settings. Ensure to fill `App Callback URL` field with `http://<your hostname>/complete/vimeo/`
- Fill in the **Client Id** and **Client Secret** values in your settings:

```
SOCIAL_AUTH_VIMEO_KEY = ''
SOCIAL_AUTH_VIMEO_SECRET = ''
```

- Specify scopes with:

```
SOCIAL_AUTH_VIMEO_SCOPE = [...]
```

- Add the backend to `AUTHENTICATION_BACKENDS`:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.vimeo.VimeoOAuth1',
    ...
)
```

VK.com (former Vkontakte)

VK.com (former Vkontakte) auth service support.

OAuth2

VK.com uses OAuth2 for Authentication.

- Register a new application at the VK.com API,
- fill `Application Id` and `Application Secret` values in the settings:

```
SOCIAL_AUTH_VK_OAUTH2_KEY = ''
SOCIAL_AUTH_VK_OAUTH2_SECRET = ''
```

- Add `'social_core.backends.vk.VKOAuth2'` into your `SOCIAL_AUTH_AUTHENTICATION_BACKENDS`.
- Then you can start using `/login/vk-oauth2` in your link href.
- Also it's possible to define extra permissions with:

```
SOCIAL_AUTH_VK_OAUTH2_SCOPE = [...]
```

See the VK.com list of permissions.

OAuth2 Application

To support OAuth2 authentication for VK.com applications:

- Create your IFrame application at VK.com.
- In application settings specify your IFrame URL `mysite.com/vk` (current default).
- Fill `Application ID` and `Application Secret` settings:

```
SOCIAL_AUTH_VK_APP_KEY = ''
SOCIAL_AUTH_VK_APP_SECRET = ''
```

- Fill `user_mode`:

```
SOCIAL_AUTH_VK_APP_USER_MODE = 2
```

Possible values:

  - `0`: there will be no check whether a user connected to your application or not
  - `1`: `python-social-auth` will check `is_app_user` parameter VK.com sends when user opens application page one time
  - `2`: (safest) `python-social-auth` will check status of user interactively (useful when you have interactive authentication via AJAX)

- Add a snippet similar to this into your login template:

```
<script src="http://vk.com/js/api/xd_connection.js?2" type="text/javascript"></
↪script>
<script type="text/javascript">
    VK.init(function() {
            VK.addCallback("onApplicationAdded", requestRights);
            VK.addCallback("onSettingsChanged", onSettingsChanged);
        }
    );

    function startConnect() {
        VK.callMethod('showInstallBox');
    }

    function requestRights() {
        VK.callMethod('showSettingsBox', 1 + 2); // 1+2 is just an example
    }

    function onSettingsChanged(settings) {
        window.location.reload();
    }
</script>
<a href="#" onclick="startConnect(); return false;">Click to authenticate</a>
```

To test, launch the server using `sudo ./manage.py mysite.com:80` for browser to be able to load it when VK.com calls IFrame URL. Open your VK.com application page via http://vk.com/app<app_id>. Now you are able to connect to application and login automatically after connection when visiting application page.

For more details see authentication for VK.com applications

OpenAPI

You can also use VK.com's own OpenAPI to log in, but you need to provide HTML template with JavaScript code to authenticate, check below for an example.

- Get an OpenAPI App Id and add it to the settings:

```
SOCIAL_AUTH_VK_OPENAPI_ID = ''
```

  This app id will be passed to the template as `VK_APP_ID`.

Snippet example:

```
<script src="http://vk.com/js/api/openapi.js" type="text/javascript"></script>
<script type="text/javascript">
    var vkAppId = {{ VK_APP_ID|default:"null" }};
    if (vkAppId) {
        VK.init({ apiId: vkAppId });
    }
    function authVK () {
        if (!vkAppId) {
            alert ("Please specify VK.com APP ID in your local settings file");
            return false;
        }
        VK.Auth.login(function(response) {
            var params = "";
            if (response.session) {
```

```
                params = "first_name=" + encodeURI(response.session.user.first_name)␣
↪+ "&last_name=" + encodeURI(response.session.user.last_name);
                params += "&nickname=" + encodeURI(response.session.user.nickname) +
↪"&id=" + encodeURI(response.session.user.id);
            }
            window.location = "{{ VK_COMPLETE_URL }}?" + params;
        });
        return false;
    }
</script>
<a href="javascript:void(0);" onclick="authVK();">Click to authorize</a>
```

Weibo OAuth

Weibo OAuth 2.0 workflow.

- Register a new application at Weibo.
- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_WEIBO_KEY = ''
SOCIAL_AUTH_WEIBO_SECRET = ''
```

By default `account id`, `profile_image_url` and `gender` are stored in extra_data field.

The user name is used by default to build the user instance `username`, sometimes this contains non-ASCII characters which might not be desirable for the website. To avoid this issue it's possible to use the Weibo `domain` which will be inside the ASCII range by defining this setting:

```
SOCIAL_AUTH_WEIBO_DOMAIN_AS_USERNAME = True
```

Withings

Withings uses OAuth v1 for Authentication.

- Register a new application at the Withings API, and
- fill `Client ID` and `Client Secret` from withings.com values in the settings:

```
SOCIAL_AUTH_WITHINGS_KEY = ''
SOCIAL_AUTH_WITHINGS_SECRET = ''
```

Wunderlist

Wunderlist uses OAuth v2 for Authentication.

- Register a new application at Wunderlist Developer Portal, and
- fill `Client Id` and `Client Secret` values in the settings:

```
SOCIAL_AUTH_WUNDERLIST_KEY = ''
SOCIAL_AUTH_WUNDERLIST_SECRET = ''
```

XING

XING uses OAuth1 for their auth mechanism, in order to enable the backend follow:

- Register a new application at XING Apps Dashboard,
- Fill **Consumer Key** and **Consumer Secret** values:

```
SOCIAL_AUTH_XING_KEY = ''
SOCIAL_AUTH_XING_SECRET = ''
```

Yahoo

Yahoo supports OpenId and OAuth2 for their auth flow.

Yahoo OpenId

OpenId doesn't require any particular configuration beside enabling the backend in the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.yahoo.YahooOpenId',
    ...
)
```

Yahoo OAuth2

OAuth 2.0 workflow, useful if you are planning to use Yahoo's API.

- Register a new application at Yahoo Developer Center, set your app domain and configure scopes (they can't be overriden by application).
- Fill `Consumer Key` and `Consumer Secret` values in the settings:

```
SOCIAL_AUTH_YAHOO_OAUTH2_KEY = ''
SOCIAL_AUTH_YAHOO_OAUTH2_SECRET = ''
```

Yammer

Yammer users OAuth2 for their auth mechanism, this application supports Yammer OAuth2 in production and staging modes.

Production Mode

In order to enable the backend, follow:

- Register an application at Client Applications, set the `Redirect URI` to `http://<your hostname>/complete/yammer/`
- Fill **Client Key** and **Client Secret** settings:

```
SOCIAL_AUTH_YAMMER_KEY = '...'
SOCIAL_AUTH_YAMMER_SECRET = '...'
```

Staging Mode

Staging mode is configured the same as `Production Mode`, but settings are prefixed with:

```
SOCIAL_AUTH_YAMMER_STAGING_*
```

Zotero

Zotero implements OAuth1 as their authentication mechanism for their Web API v3.

1. Go to the Zotero app registration page to register your application.
2. Fill the **Client ID** and **Client Secret** in your project settings:

```
SOCIAL_AUTH_ZOTERO_KEY = '...'
SOCIAL_AUTH_ZOTERO_SECRET = '...'
```

3. Enable the backend:

```
SOCIAL_AUTH_AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.zotero.ZoteroOAuth',
    ...
)
```

Further documentation at Zotero Web API v3 page.

CHAPTER 10

Beginners Guide

This is an attempt to bring together a number of concepts in python-social-auth (psa) so that you will understand how it fits into your system. This definitely has a Django flavor to it (because that's how I learned it).

10.1 Understanding PSA URLs

If you have not seen namespaced URLs before, you are about to be introduced. When you add the PSA entry to your urls.py, it looks like this:

```
url(r'', include('social_django.urls', namespace='social'))
```

that "namespace" part on the end is what keeps the names in the PSA-world from colliding with the names in your app, or other 3rd-party apps. So your login link will look like this:

```
<a href="{% url 'social:begin' 'provider-name' %}">Login</a>
```

(See how "social" in the URL mapping matches the value of "namespace" in the urls.py entry?)

10.2 Understanding Backends

PSA implements a lot of backends. Find the entry in the docs for your backend, and if it's there, follow the steps to enable it, which come down to

1. Set up SOCIAL_AUTH_{backend} variables in settings.py. (The settings vary, based on the backends)
2. Adding your backend to AUTHENTICATION_BACKENDS in settings.py.

If you need to implement a different backend (for instance, let's say you want to use Intuit's OpenID), you can subclass the nearest one and override the "name" attribute:

```
from social_core.backends.open_id import OpenIDAuth

class IntuitOpenID(OpenIDAuth):
    name = 'intuit'
```

And then add your new backend to AUTHENTICATION_BACKENDS in settings.py.

A couple notes about the pipeline:

The standard pipeline does not log the user in until after the pipeline has completed. So if you get a value in the user key of the accumulative dictionary, that implies that the user was logged in when the process started.

10.3 Understanding the Pipeline

Reversing a URL like `{% url 'social:begin' 'github' %}` will give you a url like:

```
http://example.com/login/github
```

And clicking on that link will cause the "pipeline" to be started. The pipeline is a list of functions that build up data about the user as we go through the steps of the authentication process. (If you really want to understand the pipeline, look at the source in `social/backends/base.py`, and see the `run_pipeline()` function in `BaseAuth`.)

The design contract for each function in the pipeline is:

1. The pipeline starts with a four-item dictionary (the accumulative dictionary) which is updated with the results of each function in the pipeline. The initial four values are:

   `strategy` contains a strategy object

   `backend` contains the backend being used during this pipeline run

   `request` contains a dictionary of the request keys. Note to Django users – this is not an HttpRequest object, it is actually the results of `request.REQUEST`.

   `details` which is an empty dict.

2. If the function returns a dictionary or something False-ish, add the contents of the dictionary to an accumulative dictionary (called `out` in `run_pipeline`), and call the next step in the pipeline with the accumulative dictionary.

3. If something else is returned (for example, a subclass of `HttpResponse`), then return that to the browser.

4. If the pipeline completes, *THEN* the user is authenticated (logged in). So if you are finding an authenticated user object while the pipeline is running, that means that the user was logged in when the pipeline started.

There is one pipeline for your site as a whole – if you have backend-specific logic, you have to make your pipeline steps smart enough to skip the step if it is not relevant. This is as simple as:

```
def my_custom_step(strategy, backend, request, details, *args, **kwargs):
    if backend_name != 'my_custom_backend':
        return
    # otherwise, do the special steps for your custom backend
```

10.4 Interrupting the Pipeline (and communicating with views)

Let's say you want to add a custom step in the pipeline – you want the user to establish a password so that they can come directly to your site in the future. We can do that with the *@partial* decorator, which tells the pipeline to keep

track of where it is so that it can be restarted.

The first thing we need to do is set up a way for our views to communicate with the pipeline. That is done by adding a value to the settings file to tell us which values should be passed back and forth between the session and the pipeline:

```
SOCIAL_AUTH_FIELDS_STORED_IN_SESSION = ['local_password',]
```

In our pipeline code, we would have:

```
from django.shortcuts import redirect
from django.contrib.auth.models import User
from social_core.pipeline.partial import partial

# partial says "we may interrupt, but we will come back here again"
@partial
def collect_password(strategy, backend, request, details, *args, **kwargs):
    # session 'local_password' is set by the pipeline infrastructure
    # because it exists in FIELDS_STORED_IN_SESSION
    local_password = strategy.session_get('local_password', None)
    if not local_password:
        # if we return something besides a dict or None, then that is
        # returned to the user -- in this case we will redirect to a
        # view that can be used to get a password
        return redirect("myapp.views.collect_password")

    # grab the user object from the database (remember that they may
    # not be logged in yet) and set their password.  (Assumes that the
    # email address was captured in an earlier step.)
    user = User.objects.get(email=kwargs['email'])
    user.set_password(local_password)
    user.save()

    # continue the pipeline
    return
```

In our view code, we would have something like:

```
class PasswordForm(forms.Form):
    secret_word = forms.CharField(max_length=10)

def get_user_password(request):
    if request.method == 'POST':
        form = PasswordForm(request.POST)
        if form.is_valid():
            # because of FIELDS_STORED_IN_SESSION, this will get copied
            # to the request dictionary when the pipeline is resumed
            request.session['local_password'] = form.cleaned_data['secret_word']

            # once we have the password stashed in the session, we can
            # tell the pipeline to resume by using the "complete" endpoint
            return redirect(reverse('social:complete', args=("backend_name,")))
    else:
        form = PasswordForm()

    return render(request, "password_form.html")
```

Note that the `social:complete` will re-enter the pipeline with the same function that interrupted it (in this case, collect_password).

CHAPTER 11

Disconnect and Logging Out

It's a common misconception that the `disconnect` action is the same as logging the user out, but this is not the case.

`Disconnect` is the way that your users can ask your project to "forget about my account". This implies removing the `UserSocialAuth` instance that was created, this also implies that the user won't be able to login back into your site with the social account. Instead the action will be a signup, a new user instance will be created, not related to the previous one.

Logging out is just a way to say "forget my current session", and usually implies removing cookies, invalidating a session hash, etc. The many frameworks have their own ways to logout an account (Django has `django.contrib.auth.logout`), `flask-login` has it's own way too with logout_user().

Since disconnecting a social account means that the user won't be able to log back in with that social provider into the same user, python-social-auth will check that the user account is in a valid state for disconnection (it has at least one more social account associated, or a password, etc). This behavior can be overridden by changing the Disconnection Pipeline.

CHAPTER 12

Testing python-social-auth

Testing the application is fair simple, just met the dependencies and run the testing suite.

The testing suite uses HTTPretty to mock server responses, it's not a live test against the providers API, to do it that way, a browser and a tool like Selenium are needed, that's slow, prone to errors on some cases, and some of the application examples must be running to perform the testing. Plus real Key and Secret pairs, in the end it's a mess to test functionality which is the real point.

By mocking the server responses, we can test the backends functionality (and other areas too) easily and quick.

12.1 Installing dependencies

Go to the tests directory and install the dependencies listed in the requirements.txt. Then run with `nosetests` command, or with the `run_tests.sh` script.

12.2 Tox

You can use tox to test compatibility against all supported Python versions:

```
$ pip install tox  # if not present
$ tox
```

12.3 Pending

At the moment only OAuth1, OAuth2 and OpenId backends are being tested, and just login and partial pipeline features are covered by the test. There's still a lot to work on, like:

- Frameworks support

CHAPTER 13

Use Cases

Some miscellaneous options and use cases for python-social-auth.

13.1 Return the user to the original page

There's a common scenario to return the user back to the original page from where they requested to login. For that purpose, the usual `next` query-string argument is used. The value of this parameter will be stored in the session and later used to redirect the user when login was successful.

In order to use it, just define it with your link. For instance, when using Django:

```
<a href="{% url 'social:begin' 'facebook' %}?next={{ request.path }}">Login with␣
↪Facebook</a>
```

13.2 Pass custom GET/POST parameters and retrieve them on authentication

In some cases, you might need to send data over the URL, and retrieve it while processing the after-effect. For example, for conditionally executing code in custom pipelines.

In such cases, add it to `FIELDS_STORED_IN_SESSION`.

In your settings:

```
FIELDS_STORED_IN_SESSION = ['key']
```

In template:

```
<a href="{% url 'social:begin' 'facebook' %}?key={{ value }}">Login with Facebook</a>
```

In your custom pipeline, retrieve it using:

```
strategy.session_get('key')
```

13.3 Retrieve Google+ Friends

Google provides a People API endpoint to retrieve the people in your circles on Google+. In order to access that API first we need to define the needed scope:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE = [
    'https://www.googleapis.com/auth/plus.login'
]
```

Once we have the `access token` we can call the API like this:

```
import requests

user = User.objects.get(...)
social = user.social_auth.get(provider='google-oauth2')
response = requests.get(
    'https://www.googleapis.com/plus/v1/people/me/people/visible',
    params={'access_token': social.extra_data['access_token']}
)
friends = response.json()['items']
```

13.4 Associate users by email

Sometimes it's desirable that social accounts are automatically associated if the email already matches a user account.

For example, if a user signed up with his Facebook account, then logged out and next time tries to use Google OAuth2 to login, it could be nice (if both social sites have the same email address configured) that the user gets into his initial account created by Facebook backend.

This scenario is possible by enabling the `associate_by_email` pipeline function, like this:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.social_auth.associate_by_email',  # <--- enable this one
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

This feature is disabled by default because it's not 100% secure to automate this process with all the backends. Not all the providers will validate your email account and others users could take advantage of that.

Take for instance User A registered in your site with the email `foo@bar.com`. Then a malicious user registers into another provider that doesn't validate his email with that same account. Finally this user will turn to your site (which supports that provider) and sign up to it, since the email is the same, the malicious user will take control over the User A account.

13.5 Signup by OAuth access_token

It's a common scenario that mobile applications will use an SDK to signup a user within the app, but that signup won't be reflected by python-social-auth unless the corresponding database entries are created. In order to do so, it's possible to create a view / route that creates those entries by a given `access_token`. Take the following code for instance (the code follows Django conventions, but versions for others frameworks can be implemented easily):

```
from django.contrib.auth import login

from social_django.utils import psa

# Define an URL entry to point to this view, call it passing the
# access_token parameter like ?access_token=<token>. The URL entry must
# contain the backend, like this:
#
#   url(r'^register-by-token/(?P<backend>[^/]+)/$',
#       'register_by_access_token')

@psa('social:complete')
def register_by_access_token(request, backend):
    # This view expects an access_token GET parameter, if it's needed,
    # request.backend and request.strategy will be loaded with the current
    # backend and strategy.
    token = request.GET.get('access_token')
    user = request.backend.do_auth(request.GET.get('access_token'))
    if user:
        login(request, user)
        return 'OK'
    else:
        return 'ERROR'
```

The snippet above is quite simple, it doesn't return JSON and usually this call will be done by AJAX. It doesn't return the user information, but that's something that can be extended and filled to suit the project where it's going to be used.

Note: when dealing with `OAuth1`, the `access_token` is actually a query-string composed by `oauth_token` and `oauth_token_secret`, python-social-auth expects this to be a `dict` with those keys, but if an string is detected, it will treat it as a query string in the form `oauth_token=123&oauth_token_secret=456`.

13.6 Multiple scopes per provider

At the moment python-social-auth doesn't provide a method to define multiple scopes for single backend, this is usually desired since it's recommended to ask the user for the minimum scope possible and increase the access when it's really needed. It's possible to add a new backend extending the original one to accomplish that behavior. There are two ways to do it.

1. Overriding `get_scope()` method:

```
from social_core.backends.facebook import FacebookOAuth2


class CustomFacebookOAuth2(FacebookOauth2):
    def get_scope(self):
        scope = super(CustomFacebookOAuth2, self).get_scope()
        if self.data.get('extrascope'):
            scope = scope + [('foo', 'bar')]
        return scope
```

This method is quite simple, it overrides the method that returns the scope value in a backend (`get_scope()`) and adds extra values tot he list if it was indicated by a parameter in the `GET` or `POST` data (`self.data`).

Put this new backend in some place in your project and replace the original `FacebookOAuth2` in `AUTHENTICATION_BACKENDS` with this new version.

When overriding this method, take into account that the default output the base class for `get_scope()` is the raw value from the settings (whatever they are defined), doing this will actually update the value in your settings for all the users:

```
scope = super(CustomFacebookOAuth2, self).get_scope()
scope += ['foo', 'bar']
```

Instead do it like this:

```
scope = super(CustomFacebookOAuth2, self).get_scope()
scope = scope + ['foo', 'bar']
```

2. It's possible to do the same by defining a second backend which extends from the original but overrides the name, this will imply new URLs and also new settings for the new backend (since the name is used to build the settings names), it also implies a new application in the provider since not all providers give you the option of defining multiple redirect URLs. To do it just add a backend like:

```
from social_core.backends.facebook import FacebookOAuth2


class CustomFacebookOAuth2(FacebookOauth2):
    name = 'facebook-custom'
```

Put this new backend in some place in your project keeping the original `FacebookOAuth2` in `AUTHENTICATION_BACKENDS`. Now a new set of URLs will be functional:

```
/login/facebook-custom
/complete/facebook-custom
/disconnect/facebook-custom
```

And also a new set of settings:

```
SOCIAL_AUTH_FACEBOOK_CUSTOM_KEY = '...'
SOCIAL_AUTH_FACEBOOK_CUSTOM_SECRET = '...'
SOCIAL_AUTH_FACEBOOK_CUSTOM_SCOPE = [...]
```

When the extra permissions are needed, just redirect the user to `/login/facebook-custom` and then get the social auth entry for this new backend with `user.social_auth.get(provider='facebook-custom')` and use the `access_token` in it.

13.7 Enable a user to choose a username from his World of Warcraft characters

If you want to register new users on your site via battle.net, you can enable these users to choose a username from their own World-of-Warcraft characters. To do this, use the `battlenet-oauth2` backend along with a small form to choose the username.

The form is rendered via a partial pipeline item like this:

```
@partial
def pick_character_name(backend, details, response, is_new=False, *args, **kwargs):
    if backend.name == 'battlenet-oauth2' and is_new:
        data = backend.strategy.request_data()
        if data.get('character_name') is None:
            # New user and didn't pick a character name yet, so we render
            # and send a form to pick one. The form must do a POST/GET
            # request to the same URL (/complete/battlenet-oauth2/). In this
            # example we expect the user option under the key:
            #   character_name
            # you have to filter the result list according to your needs.
            # In this example, only guild members are allowed to sign up.
            char_list = [
                c['name'] for c in backend.get_characters(response.get('access_token
↪'))
                    if 'guild' in c and c['guild'] == '<guild name>'
            ]
            return render_to_response('pick_character_form.html', {'charlist': char_
↪list, })
        else:
            # The user selected a character name
            return {'username': data.get('character_name')}
```

Don't forget to add the partial to the pipeline:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'path.to.pick_character_name',
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

It needs to be somewhere before create_user because the partial will change the username according to the users choice.

13.8 Re-prompt Google OAuth2 users to refresh the `refresh_token`

A `refresh_token` also expire, a `refresh_token` can be lost, but they can also be refreshed (or re-fetched) if you ask to Google the right way. In order to do so, set this setting:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_AUTH_EXTRA_ARGUMENTS = {
    'access_type': 'offline',
    'approval_prompt': 'auto'
}
```

Then link the users to `/login/google-oauth2?approval_prompt=force`. If you want to refresh the `refresh_token` only on those users that don't have it, do it with a pipeline function:

```
def redirect_if_no_refresh_token(backend, response, social, *args, **kwargs):
    if backend.name == 'google-oauth2' and social and \
       response.get('refresh_token') is None and \
       social.extra_data.get('refresh_token') is None:
        return redirect('/login/google-oauth2?approval_prompt=force')
```

Set this pipeline after `social_user`:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'path.to.redirect_if_no_refresh_token',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

CHAPTER 14

Thanks

python-social-auth is the result of almost 3 years of development done on django-social-auth which is the result of my initial work and the thousands lines of code contributed by so many developers that took time to work on improvements, report bugs and hunt them down to propose a fix. So, here is a big list of users that helped to build this library (if somebody is missed let me know and I'll update the list):

- kjoconnor
- krvss
- estebistec
- mrmch
- uruz
- maraujop
- bacher09
- dokterbob
- hassek
- andrusha
- vicalloy
- caioariede
- danielgtaylor
- stephenmcd
- gugu
- yrik
- dhendo
- yekibud
- tmackenzie

- LuanP
- jezdez
- serdardalgic
- Jolmberg
- ChrisCooper
- marselester
- eshellman
- micrypt
- revolunet
- dasevilla
- seansay
- hepochen
- gibuloto
- crodjer
- sidmitra
- ryr
- inve1
- mback2k
- hannesstruss
- NorthIsUp
- tonyxiao
- dhepper
- Troytft
- gardaud
- oinopion
- gameguy43
- vinigracindo
- syabro
- bashmish
- ggreer
- avillavi
- r4vi
- roderyc
- daonb
- slon7
- JasonGiedymin

- tymofij
- Cassus
- martey
- t0m
- johnthedebs
- jammons
- stefanw
- maxgrosse
- mattucf
- tadeo
- haxoza
- bradbeattie
- henward0
- bernardokyotoku
- czpython
- glasscube42
- assiotis
- dbaxa
- JasonSanford
- originell
- cihann
- niftynei
- mikesun
- 1st
- betonimig
- ozexpert
- stephenLee
- julianvargasalvarez
- youngrok
- garrypolley
- amirouche
- fmoga
- pydanny
- pygeek
- dgouldin
- kotslon

- kirkchris
- barracel
- sayar
- kulbir
- Morgul
- spstpl
- bluszcz
- vbsteven
- sbassi
- aspcanada
- browniebroke

CHAPTER 15

Copyrights and Licence

`python-social-auth` is protected by BSD licence. Check the LICENCE for details.

The base work was derived from django-social-auth work and copyrighted too, check django-social-auth LICENCE for details:

CHAPTER 16

Indices and Tables

- genindex
- modindex
- search