Flask-Social Documentation

Release 1.6.2

Matt Wright

Sep 27, 2017

Contents

Master

Develop

Oauth provider login and APIs for use with Flask-Security

CHAPTER 1

Resources

- Documentation
- Issue Tracker
- Code
- Development Version
- Example Application

CHAPTER 2

Contents

CHAPTER 3

Overview

Flask-Social sets up endpoints for your app to make it easy for you to let your users connect and/or login using Facebook and Twitter. Flask-Social persists the connection information and allows you to get a configured instance of an API object with your user's token so you can make API calls on behalf of them. Currently Facebook, Twitter, foursquare and Google are supported out of the box as long as you install the appropriate API library.

CHAPTER 4

Installation

First, install Flask-Social:

```
$ mkvirtualenv app-name
$ pip install Flask-Social
```

Then install your datastore requirement.

SQLAlchemy:

```
$ pip install Flask-SQLAlchemy
```

MongoEngine:

```
$ pip install https://github.com/sbook/flask-mongoengine/tarball/master
```

Then install your provider API libraries.

Facebook:

```
$ pip install http://github.com/pythonforfacebook/facebook-sdk/tarball/master
```

Twitter:

```
$ pip install python-twitter
```

foursquare:

```
$ pip install foursquare
```

Google:

```
$ pip install oauth2client google-api-python-client
```

CHAPTER 5

Getting Started

If you plan on allowing your users to connect with Facebook or Twitter, the first thing you'll want to do is register an application with either service provider:

- Facebook
- Twitter
- foursquare

Bear in mind that Flask-Social requires Flask-Security. It would be a good idea to review the documentation for Flask-Security before moving on here as it assumes you have knowledge and a working Flask-Security app already.

Configuration

After you register your application(s) you'll need to configure your Flask app with the consumer key and secret. When dealing with Facebook, the consumer key is also referred to as the App ID/API Key. The following is an example of how to configure your application with your provider's application values

Twitter:

```
app.config['SOCIAL_TWITTER'] = {
    'consumer_key': 'twitter consumer key',
    'consumer_secret': 'twitter consumer secret'
}
```

Facebook:

```
app.config['SOCIAL_FACEBOOK'] = {
    'consumer_key': 'facebook app id',
    'consumer_secret': 'facebook app secret'
}
```

foursquare:

```
app.config['SOCIAL_FOURSQUARE'] = {
    'consumer_key': 'client id',
    'consumer_secret': 'client secret'
}
```

Google:

```
app.config['SOCIAL_GOOGLE'] = {
    'consumer_key': 'xxxx',
    'consumer_secret': 'xxxx'
}
```

Next you'll want to setup the *Social* extension and give it an instance of your datastore. In the following code the post login page is set to a hypothetical profile page instead of Flask-Security's default of the root index:

```
# ... other required imports ...
from flask.ext.social import Social
from flask.ext.social.datastore import SQLAlchemyConnectionDatastore

# ... create the app ...

app.config['SECURITY_POST_LOGIN'] = '/profile'

db = SQLAlchemy(app)

# ... define user and role models ...

class Connection(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    provider_id = db.Column(db.String(255))
    provider_user_id = db.Column(db.String(255))
    access_token = db.Column(db.String(255))
    secret = db.Column(db.String(255))
    display_name = db.Column(db.String(255))
    profile_url = db.Column(db.String(512))
    image_url = db.Column(db.String(512))
    rank = db.Column(db.Integer)

Security(app, SQLAlchemyUserDatastore(db, User, Role))
Social(app, SQLAlchemyConnectionDatastore(db, Connection))
```

Connecting to Providers

In order to let users connect their Facebook or Twitter accounts you'll want to add a mechanism on the profile page to do so. First the view method:

```
@app.route('/profile')
@login_required
def profile():
    return render_template(
        'profile.html',
        content='Profile Page',
        twitter_conn=social.twitter.get_connection(),
```

```
        facebook_conn=social.facebook.get_connection(),
        foursquare_conn=social.foursquare.get_connection())
```

You should notice the mechanism for retreiving the current user's connection with each service provider. If a connection is not found, the value will be *None*. Now lets take a look at the profile template:

```
{% macro show_provider_button(provider_id, display_name, conn) %}
    {% if conn %}
    <form action="{{ url_for('social.remove_connection', provider_id=conn.provider_id,
↪ provider_user_id=conn.provider_user_id) }}" method="DELETE">
      <input type="submit" value="Disconnect {{ display_name }}" />
    </form>
    {% else %}
    <form action="{{ url_for('social.connect', provider_id=provider_id) }}" method=
↪"POST">
      <input type="submit" value="Connect {{ display_name }}" />
    </form>
    {% endif %}
{% endmacro %}

{{ show_provider_button('twitter', 'Twitter', twitter_conn) }}

{{ show_provider_button('facebook', 'Facebook', facebook_conn) }}

{{ show_provider_button('foursquare', 'foursquare', foursquare_conn) }}
```

In the above template code a form is displayed depending on if a connection for the current user exists or not. If the connection exists a disconnect button is displayed and if it doesn't exist a connect button is displayed. Clicking the connect button will initiate the OAuth flow with the given provider, allowing the user to authorize the application and return a token and/or secret to be used when configuring an API instance.

Logging In

If a user has a connection established to a service provider then it is possible for them to login via the provider. A login form would look like the following:

```
<form action="{{ url_for('security.authenticate') }}" method="POST" name="login_form">
  {{ form.hidden_tag() }}
  {{ form.username.label }} {{ form.username }}<br/>
  {{ form.password.label }} {{ form.password }}<br/>
  {{ form.remember.label }} {{ form.remember }}<br/>
  {{ form.submit }}
</form>

{% macro social_login(provider_id, display_name) %}
  <form action="{{ url_for('social.login', provider_id=provider_id) }}" method="POST">
    <input type="submit" value="Login with {{ display_name }}" />
  </form>
{% endmacro %}

{{ social_login('twitter', 'Twitter' )}}

{{ social_login('facebook', 'Facebook' )}}

{{ social_login('foursquare', 'foursquare' )}}
```

In the above template code you'll notice the regular username and password login form and forms for the user to login via Twitter, Facebook, and foursquare. If the user has an existing connection with the provider they will automatically be logged in without having to enter their username or password.

Provider API's

Flask Social is opinionated and uses available Python libraries when possible to interact with the API's of the stock providers. This means that you'll need to install the appropriate library for this functionality to work.

Configured instances of an API client are available via the *get_api* method of the provider instance. For example, lets say you wany to post the current user's Twitter feed:

```
social = Social(...)

@app.route('/profile')
def profile():
    twitter_api = social.twitter.get_api()
    twitter_api.PostUpdate('hello from my Flask app!')
```

CHAPTER 6

Configuration Values

- SOCIAL_URL_PREFIX: Specifies the URL prefix for the Social blueprint.
- SOCIAL_APP_URL: The URL your application is registered under with a service provider.
- SOCIAL_CONNECT_ALLOW_REDIRECT: The URL to redirect to after a user successfully authorizes a connection with a service provider.
- SOCIAL_CONNECT_DENY_REDIRECT: The URL to redirect to when a user denies the connection request with a service provider.
- SOCIAL_FLASH_MESSAGES: Specifies wether or not to flash messages during connection and login requests.
- SOCIAL_POST_OAUTH_CONNECT_SESSION_KEY: Specifies the session key to use when looking for a redirect value after a connection is made.
- SOCIAL_POST_OAUTH_LOGIN_SESSION_KEY: Specifis the session key to use when looking for a redirect value after a login is completed.

CHAPTER 7

API

Signals

See the Flask documentation on signals for information on how to use these signals in your code.

`connection_created`
: Sent when a user successfully authorizes a connection with a provider provider. In addition to the app (which is the sender), it is passed *user*, which is the current user and *connection* which is the connection that was created

`connection_failed`
: Sent when a user attempts to authorize a connection with a provider but it fails because it already exists. In addition to the app (which is the sender), it is passed *user*, which is the current user

`connection_removed`
: Sent when a user removes a connection to a provider. In addition to the app (which is the sender), it is passed *user*, which is the current user and *provider_id* which is the ID of the provider that was removed

`login_failed`
: Sent when a login attempt via a provider fails. In addition to the app (which is the sender), it is passed *provider* which is the service provider, and *oauth_response* which is the response returned by the provider

`login_completed`
: Sent when a login attempt via a provider fails. In addition to the app (which is the sender), it is passed *provider* which is the service provider, and *user* which is the current user

CHAPTER 8

Changelog

Flask-Social Changelog

Here you can see the full list of changes between each Flask-Social release.

Version 1.6.2

Released November 13 2013

- Changed provider.get_connection_values() to also return Full Name (if avaliable)
- Fixed bug in the connect_callback() fuction to redirect upon OAuth failure

Version 1.6.1

Released July 25 2013

- Changed google provider to use oauth2 service instead of plus service
- Fixed MongoEngine datastore to support multiple versions
- Fixed bug when a user denies access from OAuth provider
- Fixed foursquare provider image_url bug

Version 1.6.0

Released April 04 2013

- Added Peewee support
- Refactored tests
- Code cleanup

Version 1.5.5

Released March 03 2013

- Fixed configuration bug concerning deeper dictionary values
- Fixed facebook display name bug

Version 1.5.4

Released December 23 2012

- Change dependency requirement

Version 1.5.3

Released December 12 2012

- Fix small bug with default configuration

Version 1.5.2

Released December 11 2012

- Fix small bug with provider modules

Version 1.5.1

Released October 12 2012

- Change default blueprint name to 'social' to match up with 'security'
- Ensure commit is called after social login
- Abstract a bit of the connection callback for more flexibility. See the example app

Version 1.5.0

Released October 11 2012

- Updated to sync up with latest release of Flask-Security

Version 0.1.0

Released March 2012

- Initial release

Index

C

L