Flask-OAuthlib Documentation

Release 0.9.5

Hsiaoming Yang

Dec 03, 2018

Contents

Flask-OAuthlib is designed to be a replacement for Flask-OAuth. It depends on oauthlib.

The client part of Flask-OAuthlib shares the same API as Flask-OAuth, which is pretty and simple.

Warning: Please use https://github.com/lepture/authlib instead.

CHAPTER 1

Features

- Support for OAuth 1.0a, 1.0, 1.1, OAuth2 client
- Friendly API (same as Flask-OAuth)
- Direct integration with Flask
- Basic support for remote method invocation of RESTful APIs
- Support OAuth1 provider with HMAC and RSA signature
- Support OAuth2 provider with Bearer token

CHAPTER 2

User's Guide

This part of the documentation, which is mostly prose, begins with some background information about Flask-OAuthlib, then focuses on step-by-step instructions for getting the most out of Flask-OAuthlib

2.1 Introduction

Flask-OAuthlib is designed to be a replacement for Flask-OAuth. It depends on oauthlib.

2.1.1 Why

The original Flask-OAuth suffers from lack of maintenance, and oauthlib is a promising replacement for python-oauth2.

There are lots of non-standard services that claim they are oauth providers, but their APIs are always broken. While rewriting an oauth extension for Flask, I took them into consideration. Flask-OAuthlib does support these non-standard services.

Flask-OAuthlib also provides the solution for creating an oauth service. It supports both oauth1 and oauth2 (with Bearer Token).

2.1.2 import this

Flask-OAuthlib was developed with a few **PEP 20** idioms in mind:

```
>>> import this
```

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.

5. Readability counts.

All contributions to Flask-OAuthlib should keep these important rules in mind.

2.1.3 License

A large number of open source projects in Python are BSD Licensed, and Flask-OAuthlib is released under BSD License too.

2.2 Installation

This part of the documentation covers the installation of Flask-OAuthlib.

2.2.1 Pip

Installing Flask-OAuthlib is simple with pip:

```
$ pip install Flask-OAuthlib
```

2.2.2 Cheeseshop Mirror

If the Cheeseshop is down, you can also install Flask-OAuthlib from one of the mirrors. Crate.io is one of them:

```
$ pip install -i http://simple.crate.io/ Flask-OAuthlib
```

2.2.3 Get the Code

Flask-OAuthlib is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
git clone git://github.com/lepture/flask-oauthlib.git
```

Download the tarball:

```
$ curl -OL https://github.com/lepture/flask-oauthlib/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/lepture/flask-oauthlib/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

2.3 Client

Note: Please read https://docs.authlib.org/en/latest/client/frameworks.html

The client part keeps the same API as Flask-OAuth. The only changes are the imports:

```
from flask_oauthlib.client import OAuth
```

Attention: If you are testing the provider and the client locally, do not make them start listening on the same address because they will override the *session* of each other leading to strange bugs. eg: start the provider listening on *127.0.0.1:4000* and client listening on *localhost:4000* to avoid this problem.

2.3.1 OAuth1 Client

The difference between OAuth1 and OAuth2 in the configuration is `request_token_url`. In OAuth1 it is required, in OAuth2 it should be `None`.

To connect to a remote application create a `OAuth` object and register a remote application on it using the `remote_app()` method:

```
from flask_oauthlib.client import OAuth

oauth = OAuth()
the_remote_app = oauth.remote_app('the remote app',
    ...
)
```

A remote application must define several URLs required by the OAuth machinery:

- *request_token_url*

- *access_token_url*
- *authorize_url*

Additionally the application should define an issued *consumer_key* and *consumer_secret*.

You can find these values by registering your application with the remote application you want to connect with.

Additionally you can provide a *base_url* that is prefixed to *all* relative URLs used in the remote app.

For Twitter the setup would look like this:

```
twitter = oauth.remote_app('twitter',
    base_url='https://api.twitter.com/1/',
    request_token_url='https://api.twitter.com/oauth/request_token',
    access_token_url='https://api.twitter.com/oauth/access_token',
    authorize_url='https://api.twitter.com/oauth/authenticate',
    consumer_key='<your key here>',
    consumer_secret='<your secret here>'
)
```

Now that the application is created one can start using the OAuth system. One thing is missing: the tokengetter. OAuth uses a token and a secret to figure out who is connecting to the remote application. After authentication/authorization this information is passed to a function on your side and it is your responsibility to remember it.

The following rules apply:

- It's your responsibility to store that information somewhere
- That information lives for as long as the user did not revoke the access for your application on the remote application. If it was revoked and the user re-enabled the application you will get different keys, so if you store them in the database don't forget to check if they changed in the authorization callback.
- During the authorization handshake a temporary token and secret are issued. Your tokengetter is not used during that period.

For a simple test application, storing that information in the session is probably sufficient:

```
from flask import session

@twitter.tokengetter
def get_twitter_token(token=None):
    return session.get('twitter_token')
```

If the token does not exist, the function must return *None*, and otherwise return a tuple in the form `(token, secret)`. The function might also be passed a *token* parameter. This is user defined and can be used to indicate another token. Imagine for instance you want to support user and application tokens or different tokens for the same user.

The name of the token can be passed to to the `request()` function.

2.3.2 Signing in / Authorizing

To sign in with Twitter or link a user account with a remote Twitter user, simply call into `authorize()` and pass it the URL that the user should be redirected back to. For example:

```
@app.route('/login')
def login():
    return twitter.authorize(callback=url_for('oauth_authorized',
        next=request.args.get('next') or request.referrer or None))
```

If the application redirects back, the remote application can fetch all relevant information in the *oauth_authorized* function with `authorized_response()`:

```
from flask import redirect

@app.route('/oauth-authorized')
def oauth_authorized():
    next_url = request.args.get('next') or url_for('index')
    resp = twitter.authorized_response()
    if resp is None:
        flash(u'You denied the request to sign in.')
        return redirect(next_url)

    session['twitter_token'] = (
        resp['oauth_token'],
        resp['oauth_token_secret']
    )
    session['twitter_user'] = resp['screen_name']

    flash('You were signed in as %s' % resp['screen_name'])
    return redirect(next_url)
```

We store the token and the associated secret in the session so that the tokengetter can return it. Additionally, we also store the Twitter username that was sent back to us so that we can later display it to the user. In larger applications it is recommended to store satellite information in a database instead to ease debugging and more easily handle additional information associated with the user.

2.3.3 Facebook OAuth

For Facebook the flow is very similar to Twitter or other OAuth systems but there is a small difference. You're not using the *request_token_url* at all and you need to provide a scope in the *request_token_params*:

```
facebook = oauth.remote_app('facebook',
    base_url='https://graph.facebook.com/',
    request_token_url=None,
    access_token_url='/oauth/access_token',
    authorize_url='https://www.facebook.com/dialog/oauth',
    consumer_key=FACEBOOK_APP_ID,
    consumer_secret=FACEBOOK_APP_SECRET,
    request_token_params={'scope': 'email'}
)
```

Furthermore the *callback* is mandatory for the call to `authorize()` and has to match the base URL that was specified in the Facebook application control panel. For development you can set it to `localhost:5000`.

The *APP_ID* and *APP_SECRET* can be retrieved from the Facebook app control panel. If you don't have an application registered yet you can do this at facebook.com/developers.

2.3.4 Invoking Remote Methods

Now the user is signed in, but you probably want to use OAuth to call protected remote API methods and not just sign in. For that, the remote application object provides a `request()` method that can request information from an OAuth protected resource. Additionally there are shortcuts like `get()` or `post()` to request data with a certain HTTP method.

For example to create a new tweet you would call into the Twitter application as follows:

```
resp = twitter.post('statuses/update.json', data={
    'status':   'The text we want to tweet'
})
if resp.status == 403:
    flash('Your tweet was too long.')
else:
    flash('Successfully tweeted your tweet (ID: #%s)' % resp.data['id'])
```

Or to display the users' feed we can do something like this:

```
resp = twitter.get('statuses/home_timeline.json')
if resp.status == 200:
    tweets = resp.data
else:
    tweets = None
    flash('Unable to load tweets from Twitter. Maybe out of '
          'API calls or Twitter is overloaded.')
```

Flask-OAuthlib will do its best to send data encoded in the right format to the server and to decode it when it comes back. Incoming data is encoded based on the *mimetype* the server sent and is stored in the `data` attribute. For outgoing data a default of `'urlencode'` is assumed. When a different format is needed, one can specify it with the *format* parameter. The following formats are supported:

Outgoing:

- `'urlencode'` - form encoded data (*GET* as URL and *POST*/*PUT* as request body)
- `'json'` - JSON encoded data (*POST*/*PUT* as request body)

Incoming

- `'urlencode'` - stored as flat unicode dictionary
- `'json'` - decoded with JSON rules, most likely a dictionary
- `'xml'` - stored as elementtree element

Unknown incoming data is stored as a string. If outgoing data of a different format is needed, *content_type* should be specified instead and the data provided should be an encoded string.

Find the OAuth1 client example at twitter.py.

2.3.5 OAuth2 Client

Find the OAuth2 client example at github.py.

New in version 0.4.2.

Request state parameters in authorization can be a function:

```
from werkzeug import security

remote = oauth.remote_app(
    request_token_params={
        'state': lambda: security.gen_salt(10)
    }
)
```

2.3.6 Lazy Configuration

New in version 0.3.0.

When creating an open source project, we need to keep our consumer key and consumer secret secret. We usually keep them in a config file, and don't keep track of the config in the version control.

Client of Flask-OAuthlib has a mechanism for you to lazy load your configuration from your Flask config object:

```
from flask_oauthlib.client import OAuth

oauth = OAuth()
twitter = oauth.remote_app(
    'twitter',
    base_url='https://api.twitter.com/1/',
    request_token_url='https://api.twitter.com/oauth/request_token',
    access_token_url='https://api.twitter.com/oauth/access_token',
    authorize_url='https://api.twitter.com/oauth/authenticate',
    app_key='TWITTER'
)
```

At this moment, we didn't put the `consumer_key` and `consumer_secret` in the `remote_app`, instead, we set a `app_key`. It will load from Flask config by the key `TWITTER`, the configuration looks like:

```
app.config['TWITTER'] = {
    'consumer_key': 'a random string key',
    'consumer_secret': 'a random string secret',
}

oauth.init_app(app)
```

New in version 0.4.0.

Or looks like that:

```
app.config['TWITTER_CONSUMER_KEY'] = 'a random string key'
app.config['TWITTER_CONSUMER_SECRET'] = 'a random string secret'
```

Twitter can get consumer key and secret from the Flask instance now.

You can put all the configuration in `app.config` if you like, which means you can do it this way:

```
from flask_oauthlib.client import OAuth

oauth = OAuth()
twitter = oauth.remote_app(
    'twitter',
    app_key='TWITTER'
)

app.config['TWITTER'] = dict(
    consumer_key='a random key',
    consumer_secret='a random secret',
    base_url='https://api.twitter.com/1/',
    request_token_url='https://api.twitter.com/oauth/request_token',
    access_token_url='https://api.twitter.com/oauth/access_token',
    authorize_url='https://api.twitter.com/oauth/authenticate',
)
oauth.init_app(app)
```

2.3.7 Fix non-standard OAuth

There are services that claimed they are providing OAuth API, but with a little differences. Some services even return with the wrong Content Type.

This library takes all theses into consideration. Take an Chinese clone of twitter which is called weibo as the example. When you implement the authorization flow, the content type changes in the progress. Sometime it is application/json which is right. Sometime it is text/plain, which is wrong. And sometime, it didn't return anything.

We can force to parse the returned response in a specified content type:

```
from flask_oauthlib.client import OAuth

oauth = OAuth()

weibo = oauth.remote_app(
    'weibo',
    consumer_key='909122383',
    consumer_secret='2cdc60e5e9e14398c1cbdf309f2ebd3a',
    request_token_params={'scope': 'email,statuses_to_me_read'},
    base_url='https://api.weibo.com/2/',
    authorize_url='https://api.weibo.com/oauth2/authorize',
    request_token_url=None,
    access_token_method='POST',
    access_token_url='https://api.weibo.com/oauth2/access_token',

    # force to parse the response in applcation/json
    content_type='application/json',
)
```

The weibo site didn't follow the Bearer token, the acceptable header is:

```
'OAuth2 a-token-string'
```

The original behavior of Flask OAuthlib client is:

```
'Bearer a-token-string'
```

We can configure with a *pre_request* method to change the headers:

```
def change_weibo_header(uri, headers, body):
    auth = headers.get('Authorization')
    if auth:
        auth = auth.replace('Bearer', 'OAuth2')
        headers['Authorization'] = auth
    return uri, headers, body

weibo.pre_request = change_weibo_header
```

You can change uri, headers and body in the pre request.

2.4 OAuth1 Server

Note: Please read https://docs.authlib.org/en/latest/flask/oauth1.html

This part of documentation covers the tutorial of setting up an OAuth1 provider. An OAuth1 server concerns how to grant the authorization and how to protect the resource. Register an **OAuth** provider:

```
from flask_oauthlib.provider import OAuth1Provider

app = Flask(__name__)
oauth = OAuth1Provider(app)
```

Like any other Flask extensions, we can pass the application later:

```
oauth = OAuth1Provider()

def create_app():
    app = Flask(__name__)
    oauth.init_app(app)
    return app
```

To implemente the oauthorization flow, we need to understand the data model.

2.4.1 User (Resource Owner)

A user, or resource owner, is usally the registered user on your site. You design your own user model, there is not much to say.

2.4.2 Client (Application)

A client is the app which want to use the resource of a user. It is suggested that the client is registered by a user on your site, but it is not required.

The client should contain at least these information:

- client_key: A random string
- client_secret: A random string
- redirect_uris: A list of redirect uris
- default_redirect_uri: One of the redirect uris
- default_realms: Default realms/scopes of the client

But it could be better, if you implemented:

- validate_realms: A function to validate realms

An example of the data model in SQLAlchemy (SQLAlchemy is not required):

```
class Client(db.Model):
    # human readable name, not required
    name = db.Column(db.String(40))

    # human readable description, not required
    description = db.Column(db.String(400))

    # creator of the client, not required
    user_id = db.Column(db.ForeignKey('user.id'))
    # required if you need to support client credential
    user = db.relationship('User')
```

(continues on next page)

(continued from previous page)

```
    client_key = db.Column(db.String(40), primary_key=True)
    client_secret = db.Column(db.String(55), unique=True, index=True,
                              nullable=False)

    _realms = db.Column(db.Text)
    _redirect_uris = db.Column(db.Text)

    @property
    def redirect_uris(self):
        if self._redirect_uris:
            return self._redirect_uris.split()
        return []

    @property
    def default_redirect_uri(self):
        return self.redirect_uris[0]

    @property
    def default_realms(self):
        if self._realms:
            return self._realms.split()
        return []
```

2.4.3 Request Token and Verifier

Request token is designed for exchanging access token. Verifier token is designed to verify the current user. It is always suggested that you combine request token and verifier together.

The request token should contain:

- client: Client associated with this token
- token: Access token
- secret: Access token secret
- realms: Realms with this access token
- redirect_uri: A URI for redirecting

The verifier should contain:

- verifier: A random string for verifier
- user: The current user

And the all in one token example:

```
class RequestToken(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(
        db.Integer, db.ForeignKey('user.id', ondelete='CASCADE')
    )
    user = db.relationship('User')

    client_key = db.Column(
        db.String(40), db.ForeignKey('client.client_key'),
```

(continues on next page)

(continued from previous page)

```
        nullable=False,
    )
    client = db.relationship('Client')

    token = db.Column(db.String(255), index=True, unique=True)
    secret = db.Column(db.String(255), nullable=False)

    verifier = db.Column(db.String(255))

    redirect_uri = db.Column(db.Text)
    _realms = db.Column(db.Text)

    @property
    def realms(self):
        if self._realms:
            return self._realms.split()
        return []
```

Since the request token and verifier is a one-time token, it would be better to put them in a cache.

2.4.4 Timestamp and Nonce

Timestamp and nonce is a token for preventing repeating requests, it can store these information:

- client_key: The client/consure key
- timestamp: The `oauth_timestamp` parameter
- nonce: The `oauth_nonce` parameter
- request_token: Request token string, if any
- access_token: Access token string, if any

The timelife of a timestamp and nonce is 60 senconds, put it in a cache please. Here is an example in SQLAlchemy:

```
class Nonce(db.Model):
    id = db.Column(db.Integer, primary_key=True)

    timestamp = db.Column(db.Integer)
    nonce = db.Column(db.String(40))
    client_key = db.Column(
        db.String(40), db.ForeignKey('client.client_key'),
        nullable=False,
    )
    client = db.relationship('Client')
    request_token = db.Column(db.String(50))
    access_token = db.Column(db.String(50))
```

2.4.5 Access Token

An access token is the final token that could be use by the client. Client will send access token everytime when it need to access resource.

A access token requires at least these information:

- client: Client associated with this token

- user: User associated with this token
- token: Access token
- secret: Access token secret
- realms: Realms with this access token

The implementation in SQLAlchemy:

```
class AccessToken(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    client_key = db.Column(
        db.String(40), db.ForeignKey('client.client_key'),
        nullable=False,
    )
    client = db.relationship('Client')

    user_id = db.Column(
        db.Integer, db.ForeignKey('user.id'),
    )
    user = db.relationship('User')

    token = db.Column(db.String(255))
    secret = db.Column(db.String(255))

    _realms = db.Column(db.Text)

    @property
    def realms(self):
        if self._realms:
            return self._realms.split()
        return []
```

2.4.6 Configuration

The oauth provider has some built-in defaults, you can change them with Flask config:

OAUTH1_PROVIDER_ERROR_URI	The error page when there is an error, default value is `'/oauth/errors'`.
OAUTH1_PROVIDER_ERROR_ENDPOINT	You can also configure the error page uri with an endpoint name.
OAUTH1_PROVIDER_REALMS	A list of allowed realms, default is [].
OAUTH1_PROVIDER_KEY_LENGTH	A range allowed for key length, default value is (20, 30).
OAUTH1_PROVIDER_ENFORCE_SSL	If the server should be enforced through SSL. Default value is True.
OAUTH1_PROVIDER_SIGNATURE_METHODS	Allowed signature methods, default value is (SIGNATURE_HMAC, SIGNATURE_RSA).

Warning: RSA signature is not ready at this moment, you should use HMAC.

2.4.7 Implements

The implementings of authorization flow needs three handlers, one is request token handler, one is authorize handler for user to confirm the grant, the other is token handler for client to exchange access token.

Before the implementing of authorize and request/access token handler, we need to set up some getters and setter to communicate with the database.

Client getter

A client getter is required. It tells which client is sending the requests, creating the getter with decorator:

```
@oauth.clientgetter
def load_client(client_key):
    return Client.query.filter_by(client_key=client_key).first()
```

Request token & verifier getters and setters

Request token & verifier getters and setters are required. They are used in the authorization flow, implemented with decorators:

```
@oauth.grantgetter
def load_request_token(token):
    grant = RequestToken.query.filter_by(token=token).first()
    return grant

@oauth.grantsetter
def save_request_token(token, request):
    if oauth.realms:
        realms = ' '.join(request.realms)
    else:
        realms = None
    grant = RequestToken(
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        client=request.client,
        redirect_uri=request.redirect_uri,
        _realms=realms,
    )
    db.session.add(grant)
    db.session.commit()
    return grant

@oauth.verifiergetter
def load_verifier(verifier, token):
    return RequestToken.query.filter_by(verifier=verifier, token=token).first()

@oauth.verifiersetter
def save_verifier(token, verifier, *args, **kwargs):
    tok = RequestToken.query.filter_by(token=token).first()
    tok.verifier = verifier['oauth_verifier']
    tok.user = get_current_user()
    db.session.add(tok)
    db.session.commit()
    return tok
```

In the sample code, there is a `get_current_user` method, that will return the current user object, you should implement it yourself.

The `token` for `grantsetter` is a dict, that contains:

```
{
    u'oauth_token': u'arandomstringoftoken',
    u'oauth_token_secret': u'arandomstringofsecret',
    u'oauth_authorized_realms': u'email address'
}
```

And the `verifier` for `verifiersetter` is a dict too, it contains:

```
{
    u'oauth_verifier': u'Gqm3id67MdkrASOCQIAlb3XODaPlun',
    u'oauth_token': u'eTYP46AJbhp8u4LE5QMjXeItRGGoAI',
    u'resource_owner_key': u'eTYP46AJbhp8u4LE5QMjXeItRGGoAI'
}
```

Token getter and setter

Token getter and setters are required. They are used in the authorization flow and accessing resource flow. Implemented with decorators:

```
@oauth.tokengetter
def load_access_token(client_key, token, *args, **kwargs):
    t = AccessToken.query.filter_by(
            client_key=client_key, token=token).first()
    return t

@oauth.tokensetter
def save_access_token(token, request):
    tok = AccessToken(
        client=request.client,
        user=request.user,
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        _realms=token['oauth_authorized_realms'],
    )
    db.session.add(tok)
    db.session.commit()
```

The setter receives `token` and `request` parameters. The `token` is a dict, which contains:

```
{
    u'oauth_token_secret': u'H1xGH4X1ZkRAulHHdLfdFm7NR350tr',
    u'oauth_token': u'aXNlKcjkVImnTfTKj8CgFpc1XRZr6P',
    u'oauth_authorized_realms': u'email'
}
```

The `request` is an object, it contains at least a *user* and *client* objects for current flow.

Timestamp and Nonce getter and setter

Timestamp and Nonce getter and setter is required. They are used everywhere:

```
@oauth.noncegetter
def load_nonce(client_key, timestamp, nonce, request_token, access_token):
    return Nonce.query.filter_by(
```

(continues on next page)

(continued from previous page)

```
        client_key=client_key, timestamp=timestamp, nonce=nonce,
        request_token=request_token, access_token=access_token,
    ).first()

@oauth.noncesetter
def save_nonce(client_key, timestamp, nonce, request_token, access_token):
    nonce = Nonce(
        client_key=client_key,
        timestamp=timestamp,
        nonce=nonce,
        request_token=request_token,
        access_token=access_token,
    )
    db.session.add(nonce)
    db.session.commit()
    return nonce
```

Request token handler

Request token handler is a decorator for generating request token. You don't need to do much:

```
@app.route('/oauth/request_token')
@oauth.request_token_handler
def request_token():
    return {}
```

You can add more data on token response:

```
@app.route('/oauth/request_token')
@oauth.request_token_handler
def request_token():
    return {'version': '0.1.0'}
```

Authorize handler

Authorize handler is a decorator for authorize endpoint. It is suggested that you implemented it this way:

```
@app.route('/oauth/authorize', methods=['GET', 'POST'])
@require_login
@oauth.authorize_handler
def authorize(*args, **kwargs):
    if request.method == 'GET':
        client_key = kwargs.get('resource_owner_key')
        client = Client.query.filter_by(client_key=client_key).first()
        kwargs['client'] = client
        return render_template('authorize.html', **kwargs)
    confirm = request.form.get('confirm', 'no')
    return confirm == 'yes'
```

The GET request will render a page for user to confirm the grant, parameters in kwargs are:

- resource_owner_key: same as client_key
- realms: realms that this client requests

The POST request needs to return a bool value that tells whether user grantted the access or not.

Access token handler

Access token handler is a decorator for exchange access token. Client will request an access token with a request token. You don't need to do much:

```
@app.route('/oauth/access_token')
@oauth.access_token_handler
def access_token():
    return {}
```

Just like request token handler, you can add more data in access token.

2.4.8 Protect Resource

Protect the resource of a user with `require_oauth` decorator now:

```
@app.route('/api/me')
@oauth.require_oauth('email')
def me():
    user = request.oauth.user
    return jsonify(email=user.email, username=user.username)

@app.route('/api/user/<username>')
@oauth.require_oauth('email')
def user(username):
    user = User.query.filter_by(username=username).first()
    return jsonify(email=user.email, username=user.username)
```

The decorator accepts a list of realms, only the clients with the given realms can access the defined resources.

Changed in version 0.5.0.

The `request` has an additional property `oauth`, it contains at least:

- client: client model object
- realms: a list of scopes
- user: user model object
- headers: headers of the request
- body: body content of the request

2.4.9 Example for OAuth 1

Here is an example of OAuth 1 server: https://github.com/lepture/example-oauth1-server

Also read this article http://lepture.com/en/2013/create-oauth-server.

2.5 OAuth2 Server

Note: Please read https://docs.authlib.org/en/latest/flask/oauth2.html

An OAuth2 server concerns how to grant the authorization and how to protect the resource. Register an **OAuth** provider:

```
from flask_oauthlib.provider import OAuth2Provider

app = Flask(__name__)
oauth = OAuth2Provider(app)
```

Like any other Flask extensions, we can pass the application later:

```
oauth = OAuth2Provider()

def create_app():
    app = Flask(__name__)
    oauth.init_app(app)
    return app
```

To implement the authorization flow, we need to understand the data model.

2.5.1 User (Resource Owner)

A user, or resource owner, is usually the registered user on your site. You need to design your own user model.

2.5.2 Client (Application)

A client is the app which wants to use the resource of a user. It is suggested that the client is registered by a user on your site, but it is not required.

The client should contain at least these properties:

- client_id: A random string
- client_secret: A random string
- client_type: A string represents if it is *confidential*
- redirect_uris: A list of redirect uris
- default_redirect_uri: One of the redirect uris
- default_scopes: Default scopes of the client

But it could be better, if you implemented:

- allowed_grant_types: A list of grant types
- allowed_response_types: A list of response types
- validate_scopes: A function to validate scopes

Note: The value of the scope parameter is expressed as a list of space- delimited, case-sensitive strings.

via: http://tools.ietf.org/html/rfc6749#section-3.3

An example of the data model in SQLAlchemy (SQLAlchemy is not required):

```python
class Client(db.Model):
    # human readable name, not required
    name = db.Column(db.String(40))

    # human readable description, not required
    description = db.Column(db.String(400))

    # creator of the client, not required
    user_id = db.Column(db.ForeignKey('user.id'))
    # required if you need to support client credential
    user = db.relationship('User')

    client_id = db.Column(db.String(40), primary_key=True)
    client_secret = db.Column(db.String(55), unique=True, index=True,
                              nullable=False)

    # public or confidential
    is_confidential = db.Column(db.Boolean)

    _redirect_uris = db.Column(db.Text)
    _default_scopes = db.Column(db.Text)

    @property
    def client_type(self):
        if self.is_confidential:
            return 'confidential'
        return 'public'

    @property
    def redirect_uris(self):
        if self._redirect_uris:
            return self._redirect_uris.split()
        return []

    @property
    def default_redirect_uri(self):
        return self.redirect_uris[0]

    @property
    def default_scopes(self):
        if self._default_scopes:
            return self._default_scopes.split()
        return []
```

2.5.3 Grant Token

A grant token is created in the authorization flow, and will be destroyed when the authorization is finished. In this case, it would be better to store the data in a cache, which leads to better performance.

A grant token should contain at least this information:

- client_id: A random string of client_id
- code: A random string
- user: The authorization user
- scopes: A list of scope

- expires: A datetime.datetime in UTC
- redirect_uri: A URI string
- delete: A function to delete itself

Also in an SQLAlchemy model (this should be in a cache):

```
class Grant(db.Model):
    id = db.Column(db.Integer, primary_key=True)

    user_id = db.Column(
        db.Integer, db.ForeignKey('user.id', ondelete='CASCADE')
    )
    user = db.relationship('User')

    client_id = db.Column(
        db.String(40), db.ForeignKey('client.client_id'),
        nullable=False,
    )
    client = db.relationship('Client')

    code = db.Column(db.String(255), index=True, nullable=False)

    redirect_uri = db.Column(db.String(255))
    expires = db.Column(db.DateTime)

    _scopes = db.Column(db.Text)

    def delete(self):
        db.session.delete(self)
        db.session.commit()
        return self

    @property
    def scopes(self):
        if self._scopes:
            return self._scopes.split()
        return []
```

2.5.4 Bearer Token

A bearer token is the final token that could be used by the client. There are other token types, but bearer token is widely used. Flask-OAuthlib only comes with a bearer token.

A bearer token requires at least this information:

- access_token: A string token
- refresh_token: A string token
- client_id: ID of the client
- scopes: A list of scopes
- expires: A *datetime.datetime* object
- user: The user object
- delete: A function to delete itself

An example of the data model in SQLAlchemy:

```
class Token(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    client_id = db.Column(
        db.String(40), db.ForeignKey('client.client_id'),
        nullable=False,
    )
    client = db.relationship('Client')

    user_id = db.Column(
        db.Integer, db.ForeignKey('user.id')
    )
    user = db.relationship('User')

    # currently only bearer is supported
    token_type = db.Column(db.String(40))

    access_token = db.Column(db.String(255), unique=True)
    refresh_token = db.Column(db.String(255), unique=True)
    expires = db.Column(db.DateTime)
    _scopes = db.Column(db.Text)

    def delete(self):
        db.session.delete(self)
        db.session.commit()
        return self

    @property
    def scopes(self):
        if self._scopes:
            return self._scopes.split()
        return []
```

2.5.5 Configuration

The Oauth provider has some built-in defaults. You can change them with Flask config:

OAUTH2_PROVIDER_ERROR_URI	The error page when there is an error, default value is `'/oauth/errors'`.
OAUTH2_PROVIDER_ERROR_ENDPOINT	You can also configure the error page uri with an endpoint name.
OAUTH2_PROVIDER_TOKEN_EXPIRES_IN	Default Bearer token expires time, default is `3600`.

2.5.6 Implementation

The implementation of the authorization flow needs two handlers: one is the authorization handler for the user to confirm the grant, the other is the token handler for the client to exchange/refresh access tokens.

Before implementing the authorize and token handlers, we need to set up some getters and setters to communicate with the database.

Client getter

A client getter is required. It tells which client is sending the requests, creating the getter with a decorator:

```
@oauth.clientgetter
def load_client(client_id):
    return Client.query.filter_by(client_id=client_id).first()
```

Grant getter and setter

Grant getter and setter are required. They are used in the authorization flow, implemented with decorators:

```
from datetime import datetime, timedelta

@oauth.grantgetter
def load_grant(client_id, code):
    return Grant.query.filter_by(client_id=client_id, code=code).first()

@oauth.grantsetter
def save_grant(client_id, code, request, *args, **kwargs):
    # decide the expires time yourself
    expires = datetime.utcnow() + timedelta(seconds=100)
    grant = Grant(
        client_id=client_id,
        code=code['code'],
        redirect_uri=request.redirect_uri,
        _scopes=' '.join(request.scopes),
        user=get_current_user(),
        expires=expires
    )
    db.session.add(grant)
    db.session.commit()
    return grant
```

In the sample code, there is a `get_current_user` method, that will return the current user object. You should implement it yourself.

The `request` object is defined by `OAuthlib`. You can get at least this much information:

- client: client model object
- scopes: a list of scopes
- user: user model object
- redirect_uri: redirect_uri parameter
- headers: headers of the request
- body: body content of the request
- state: state parameter
- response_type: response_type paramter

Token getter and setter

Token getter and setter are required. They are used in the authorization flow and the accessing resource flow. They are implemented with decorators as follows:

```
@oauth.tokengetter
def load_token(access_token=None, refresh_token=None):
    if access_token:
        return Token.query.filter_by(access_token=access_token).first()
    elif refresh_token:
        return Token.query.filter_by(refresh_token=refresh_token).first()

from datetime import datetime, timedelta

@oauth.tokensetter
def save_token(token, request, *args, **kwargs):
    toks = Token.query.filter_by(client_id=request.client.client_id,
                                 user_id=request.user.id)
    # make sure that every client has only one token connected to a user
    for t in toks:
        db.session.delete(t)

    expires_in = token.get('expires_in')
    expires = datetime.utcnow() + timedelta(seconds=expires_in)

    tok = Token(
        access_token=token['access_token'],
        refresh_token=token['refresh_token'],
        token_type=token['token_type'],
        _scopes=token['scope'],
        expires=expires,
        client_id=request.client.client_id,
        user_id=request.user.id,
    )
    db.session.add(tok)
    db.session.commit()
    return tok
```

The getter will receive two parameters. If you don't need to support a refresh token, you can just load token by access token.

The setter receives `token` and `request` parameters. The `token` is a dict, which contains:

```
{
    u'access_token': u'6JwgO77PApxsFCU8Quz0pnL9s23016',
    u'refresh_token': u'7cYSMmBg4T7F4kwoWfUQA99J8yqjp0',
    u'token_type': u'Bearer',
    u'expires_in': 3600,
    u'scope': u'email address'
}
```

The `request` is an object like the one in grant setter.

User getter

User getter is optional. It is only required if you need password credential authorization:

```
@oauth.usergetter
def get_user(username, password, *args, **kwargs):
    user = User.query.filter_by(username=username).first()
    if user.check_password(password):
```

(continues on next page)

(continued from previous page)

```
        return user
    return None
```

Authorize handler

Authorize handler is a decorator for the authorize endpoint. It is suggested that you implemented it this way:

```
@app.route('/oauth/authorize', methods=['GET', 'POST'])
@require_login
@oauth.authorize_handler
def authorize(*args, **kwargs):
    if request.method == 'GET':
        client_id = kwargs.get('client_id')
        client = Client.query.filter_by(client_id=client_id).first()
        kwargs['client'] = client
        return render_template('oauthorize.html', **kwargs)

    confirm = request.form.get('confirm', 'no')
    return confirm == 'yes'
```

The GET request will render a page for user to confirm the grant. The parameters in kwargs are:

- client_id: id of the client
- scopes: a list of scope
- state: state parameter
- redirect_uri: redirect_uri parameter
- response_type: response_type parameter

The POST request needs to return a boolean value that tells whether user granted access or not.

There is a `@require_login` decorator in the sample code. You should implement this yourself. Here is an example by Flask documentation.

Token handler

Token handler is a decorator for exchanging/refreshing access token. You don't need to do much:

```
@app.route('/oauth/token')
@oauth.token_handler
def access_token():
    return None
```

You can add more data on the token response:

```
@app.route('/oauth/token')
@oauth.token_handler
def access_token():
    return {'version': '0.1.0'}
```

Limit the HTTP method with Flask routes, for example, only POST is allowed for exchange tokens:

```
@app.route('/oauth/token', methods=['POST'])
@oauth.token_handler
def access_token():
    return None
```

The authorization flow is finished, everything should be working now.

Note:

This token endpoint is for access token and refresh token both. But please remember that refresh token is only available for confidential client, and only available in password credential.

Revoke handler

In some cases a user may wish to revoke access given to an application and the revoke handler makes it possible for an application to programmatically revoke the access given to it. Also here you don't need to do much, allowing POST only is recommended:

```
@app.route('/oauth/revoke', methods=['POST'])
@oauth.revoke_handler
def revoke_token(): pass
```

Subclass way

If you are not satisfied with the decorator way of getters and setters, you can implement them in the subclass way:

```
class MyProvider(OAuth2Provider):
    def _clientgetter(self, client_id):
        return Client.query.filter_by(client_id=client_id).first()

    #: more getters and setters
```

Every getter and setter is started with _.

2.5.7 Protect Resource

Protect the resource of a user with `require_oauth` decorator now:

```
@app.route('/api/me')
@oauth.require_oauth('email')
def me():
    user = request.oauth.user
    return jsonify(email=user.email, username=user.username)

@app.route('/api/user/<username>')
@oauth.require_oauth('email')
def user(username):
    user = User.query.filter_by(username=username).first()
    return jsonify(email=user.email, username=user.username)
```

The decorator accepts a list of scopes and only the clients with the given scopes can access the defined resources.

Changed in version 0.5.0.

The `request` has an additional property `oauth`, which contains at least:

- client: client model object
- scopes: a list of scopes
- user: user model object
- redirect_uri: redirect_uri parameter
- headers: headers of the request
- body: body content of the request
- state: state parameter
- response_type: response_type paramter

2.5.8 Example for OAuth 2

An example server (and client) can be found in the tests folder: https://github.com/lepture/flask-oauthlib/tree/master/tests/oauth2

Other helpful resources include:

- Another example of an OAuth 2 server: https://github.com/authlib/example-oauth2-server
- An article on how to create an OAuth server: http://lepture.com/en/2013/create-oauth-server.

2.6 Additional Features

This documentation covers some additional features. They are not required, but they may be very helpful.

2.6.1 Request Hooks

Like Flask, Flask-OAuthlib has before_request and after_request hooks too. It is usually useful for setting limitation on the client request with before_request:

```
@oauth.before_request
def limit_client_request():
    from flask_oauthlib.utils import extract_params
    uri, http_method, body, headers = extract_params()
    request = oauth._create_request(uri, http_method, body, headers)

    client_id = request.client_key
    if not client_id:
        return
    client = Client.get(client_id)
    if over_limit(client):
        return abort(403)

    track_request(client)
```

And you can also modify the response with after_request:

```
@oauth.after_request
def valid_after_request(valid, request):
    if request.user in black_list:
        return False, request
    return valid, oauth
```

2.6.2 Bindings

Changed in version 0.4. Bindings are objects you can use to configure flask-oauthlib for use with various data stores. They allow you to define the required getters and setters for each data store with little effort.

SQLAlchemy OAuth2

bind_sqlalchemy() sets up getters and setters for storing the user, client, token and grant with SQLAlchemy, with some sane defaults. To use this class you'll need to create a SQLAlchemy model for each object. You can find examples of how to setup your SQLAlchemy models here: ref:*oauth2*.

You'll also need to provide another function which returns the currently logged-in user.

An example of how to use *bind_sqlalchemy()*:

```
oauth = OAuth2Provider(app)

bind_sqlalchemy(oauth, db.session, user=User, client=Client,
                token=Token, grant=Grant, current_user=current_user)
```

Any of the classes can be omitted if you wish to register the getters and setters yourself:

```
oauth = OAuth2Provider(app)

bind_sqlalchemy(oauth, db.session, user=User, client=Client,
                token=Token)

@oauth.grantgetter
def get_grant(client_id, code):
    pass

@oauth.grantsetter
def set_grant(client_id, code, request, *args, **kwargs):
    pass

# register tokensetter with oauth but keeping the tokengetter
# registered by `SQLAlchemyBinding`
# You would only do this for the token and grant since user and client
# only have getters
@oauth.tokensetter
def set_token(token, request, *args, **kwargs):
    pass
```

current_user is only used with the Grant bindings, therefore if you are going to register your own grant getter and setter you don't need to provide that function.

Grant Cache

Since the life of a Grant token is very short (usually about 100 seconds), storing it in a relational database is inefficient. The *bind_cache_grant()* allows you to more efficiently cache the grant token using Memcache, Redis, or some other caching system.

An example:

```
oauth = OAuth2Provider(app)
app.config.update({'OAUTH2_CACHE_TYPE': 'redis'})

bind_cache_grant(app, oauth, current_user)
```

- *app*: flask application
- *oauth*: OAuth2Provider instance
- *current_user*: a function that returns the current user

The configuration options are described below. The *bind_cache_grant()* will use the configuration options from *Flask-Cache* if they are set, else it will set them to the following defaults. Any configuration specific to *bind_cache_grant()* will take precedence over any *Flask-Cache* configuration that has been set.

CHAPTER 3

API Documentation

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 Developer Interface

This part of the documentation covers the interface of Flask-OAuthlib.

3.1.1 Client Reference

class flask_oauthlib.client.**OAuth**(*app=None*)

Registry for remote applications.

Parameters **app** – the app instance of Flask

Create an instance with Flask:

```
oauth = OAuth(app)
```

init_app(*app*)

Init app with Flask instance.

You can also pass the instance of Flask later:

```
oauth = OAuth()
oauth.init_app(app)
```

remote_app(*name*, *register=True*, ***kwargs*)

Registers a new remote application.

Parameters

- **name** – the name of the remote application
- **register** – whether the remote app will be registered

Find more parameters from *OAuthRemoteApp*.

class flask_oauthlib.client.**OAuthRemoteApp**(*oauth*, *name*, *base_url=None*, *request_token_url=None*, *access_token_url=None*, *authorize_url=None*, *consumer_key=None*, *consumer_secret=None*, *rsa_key=None*, *signature_method=None*, *request_token_params=None*, *request_token_method=None*, *access_token_params=None*, *access_token_method=None*, *access_token_headers=None*, *content_type=None*, *app_key=None*, *encoding='utf-8'*)

Represents a remote application.

Parameters

- **oauth** – the associated *OAuth* object
- **name** – the name of the remote application
- **base_url** – the base url for every request
- **request_token_url** – the url for requesting new tokens
- **access_token_url** – the url for token exchange
- **authorize_url** – the url for authorization
- **consumer_key** – the application specific consumer key
- **consumer_secret** – the application specific consumer secret
- **request_token_params** – an optional dictionary of parameters to forward to the request token url or authorize url depending on oauth version
- **request_token_method** – the HTTP method that should be used for the access_token_url. Default is `GET`
- **access_token_params** – an optional dictionary of parameters to forward to the access token url
- **access_token_method** – the HTTP method that should be used for the access_token_url. Default is `GET`
- **access_token_headers** – additonal headers that should be used for the access_token_url.
- **content_type** – force to parse the content with this content_type, usually used when the server didn't return the right content type.

New in version 0.3.0.

Parameters **app_key** – lazy load configuration from Flask app config with this app key

authorize(*callback=None*, *state=None*, ***kwargs*)

Returns a redirect response to the remote authorization URL with the signed callback given.

Parameters

- **callback** – a redirect url for the callback

- **state** – an optional value to embed in the OAuth request. Use this if you want to pass around application state (e.g. CSRF tokens).
- **kwargs** – add optional key/value pairs to the query string

authorized_handler(*f*)
Handles an OAuth callback.

Changed in version 0.7: @authorized_handler is deprecated in favor of authorized_response.

authorized_response(*args=None*)
Handles authorization response smartly.

delete(**args*, ***kwargs*)
Sends a `DELETE` request. Accepts the same parameters as *request()*.

get(**args*, ***kwargs*)
Sends a `GET` request. Accepts the same parameters as *request()*.

handle_oauth1_response(*args*)
Handles an oauth1 authorization response.

handle_oauth2_response(*args*)
Handles an oauth2 authorization response.

handle_unknown_response()
Handles a unknown authorization response.

patch(**args*, ***kwargs*)
Sends a `PATCH` request. Accepts the same parameters as *post()*.

post(**args*, ***kwargs*)
Sends a `POST` request. Accepts the same parameters as *request()*.

put(**args*, ***kwargs*)
Sends a `PUT` request. Accepts the same parameters as *request()*.

request(*url*, *data=None*, *headers=None*, *format='urlencoded'*, *method='GET'*, *content_type=None*, *token=None*)
Sends a request to the remote server with OAuth tokens attached.

Parameters

- **data** – the data to be sent to the server.
- **headers** – an optional dictionary of headers.
- **format** – the format for the *data*. Can be *urlencoded* for URL encoded data or *json* for JSON.
- **method** – the HTTP request method to use.
- **content_type** – an optional content type. If a content type is provided, the data is passed as it, and the *format* is ignored.
- **token** – an optional token to pass, if it is None, token will be generated by tokengetter.

tokengetter(*f*)
Register a function as token getter.

class flask_oauthlib.client.**OAuthResponse**(*resp*, *content*, *content_type=None*)

status
The status code of the response.

class flask_oauthlib.client.**OAuthException**(*message*, *type=None*, *data=None*)

3.1.2 OAuth1 Provider

class flask_oauthlib.provider.**OAuth1Provider**(*app=None*)

Provide secure services using OAuth1.

Like many other Flask extensions, there are two usage modes. One is binding the Flask app instance:

```
app = Flask(__name__)
oauth = OAuth1Provider(app)
```

The second possibility is to bind the Flask app later:

```
oauth = OAuth1Provider()

def create_app():
    app = Flask(__name__)
    oauth.init_app(app)
    return app
```

And now you can protect the resource with realms:

```
@app.route('/api/user')
@oauth.require_oauth('email', 'username')
def user():
    return jsonify(request.oauth.user)
```

access_token_handler(*f*)

Access token handler decorator.

The decorated function should return an dictionary or None as the extra credentials for creating the token response.

If you don't need to add any extra credentials, it could be as simple as:

```
@app.route('/oauth/access_token')
@oauth.access_token_handler
def access_token():
    return {}
```

after_request(*f*)

Register functions to be invoked after accessing the resource.

The function accepts `valid` and `request` as parameters, and it should return a tuple of them:

```
@oauth.after_request
def valid_after_request(valid, oauth):
    if oauth.user in black_list:
        return False, oauth
    return valid, oauth
```

authorize_handler(*f*)

Authorization handler decorator.

This decorator will sort the parameters and headers out, and pre validate everything:

```
@app.route('/oauth/authorize', methods=['GET', 'POST'])
@oauth.authorize_handler
def authorize(*args, **kwargs):
    if request.method == 'GET':
        # render a page for user to confirm the authorization
        return render_template('oauthorize.html')

    confirm = request.form.get('confirm', 'no')
    return confirm == 'yes'
```

before_request(*f*)
Register functions to be invoked before accessing the resource.

The function accepts nothing as parameters, but you can get information from *Flask.request* object. It is usually useful for setting limitation on the client request:

```
@oauth.before_request
def limit_client_request():
    client_key = request.values.get('client_key')
    if not client_key:
        return
    client = Client.get(client_key)
    if over_limit(client):
        return abort(403)

    track_request(client)
```

clientgetter(*f*)
Register a function as the client getter.

The function accepts one parameter *client_key*, and it returns a client object with at least these information:

- client_key: A random string
- client_secret: A random string
- redirect_uris: A list of redirect uris
- default_realms: Default scopes of the client

The client may contain more information, which is suggested:

- default_redirect_uri: One of the redirect uris

Implement the client getter:

```
@oauth.clientgetter
def get_client(client_key):
    client = get_client_model(client_key)
    # Client is an object
    return client
```

confirm_authorization_request()
When consumer confirm the authrozation.

error_uri
The error page URI.

When something turns error, it will redirect to this error page. You can configure the error page URI with Flask config:

```
OAUTH1_PROVIDER_ERROR_URI = '/error'
```

You can also define the error page by a named endpoint:

```
OAUTH1_PROVIDER_ERROR_ENDPOINT = 'oauth.error'
```

grantgetter(*f*)

Register a function as the request token getter.

The function accepts a *token* parameter, and it returns an request token object contains:

- client: Client associated with this token
- token: Access token
- secret: Access token secret
- realms: Realms with this access token
- redirect_uri: A URI for redirecting

Implement the token getter:

```
@oauth.tokengetter
def get_request_token(token):
    return RequestToken.get(token=token)
```

grantsetter(*f*)

Register a function as the request token setter.

The setter accepts a token and request parameters:

```
@oauth.grantsetter
def save_request_token(token, request):
    data = RequestToken(
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        client=request.client,
        redirect_uri=oauth.redirect_uri,
        realms=request.realms,
    )
    return data.save()
```

init_app(*app*)

This callback can be used to initialize an application for the oauth provider instance.

noncegetter(*f*)

Register a function as the nonce and timestamp getter.

The function accepts parameters:

- client_key: The client/consure key
- timestamp: The `oauth_timestamp` parameter
- nonce: The `oauth_nonce` parameter
- request_token: Request token string, if any
- access_token: Access token string, if any

A nonce and timestamp make each request unique. The implementation:

```
@oauth.noncegetter
def get_nonce(client_key, timestamp, nonce, request_token,
              access_token):
    return Nonce.get("...")
```

noncesetter(*f*)
Register a function as the nonce and timestamp setter.

The parameters are the same with *noncegetter()*:

```
@oauth.noncegetter
def save_nonce(client_key, timestamp, nonce, request_token,
               access_token):
    data = Nonce("...")
    return data.save()
```

The timestamp will be expired in 60s, it would be a better design if you put timestamp and nonce object in a cache.

request_token_handler(*f*)
Request token handler decorator.

The decorated function should return an dictionary or None as the extra credentials for creating the token response.

If you don't need to add any extra credentials, it could be as simple as:

```
@app.route('/oauth/request_token')
@oauth.request_token_handler
def request_token():
    return {}
```

require_oauth(**realms*, ***kwargs*)
Protect resource with specified scopes.

server
All in one endpoints. This property is created automaticly if you have implemented all the getters and setters.

tokengetter(*f*)
Register a function as the access token getter.

The function accepts *client_key* and *token* parameters, and it returns an access token object contains:

- client: Client associated with this token
- user: User associated with this token
- token: Access token
- secret: Access token secret
- realms: Realms with this access token

Implement the token getter:

```
@oauth.tokengetter
def get_access_token(client_key, token):
    return AccessToken.get(client_key=client_key, token=token)
```

tokensetter(*f*)
Register a function as the access token setter.

The setter accepts two parameters at least, one is token, the other is request:

```
@oauth.tokensetter
def save_access_token(token, request):
    access_token = AccessToken(
        client=request.client,
        user=request.user,
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        realms=token['oauth_authorized_realms'].split(' '),
    )
    return access_token.save()
```

The parameter token is a dict, that looks like:

```
{
    u'oauth_token': u'arandomstringoftoken',
    u'oauth_token_secret': u'arandomstringofsecret',
    u'oauth_authorized_realms': u'email address'
}
```

The *request* object would provide these information (at least):

```
- client: Client object associated with this token
- user: User object associated with this token
- request_token: Requst token for exchanging this access token
```

verifiergetter(*f*)

Register a function as the verifier getter.

The return verifier object should at least contain a user object which is the current user.

The implemented code looks like:

```
@oauth.verifiergetter
def load_verifier(verifier, token):
    data = Verifier.get(verifier)
    if data.request_token == token:
        # check verifier for safety
        return data
    return data
```

verifiersetter(*f*)

Register a function as the verifier setter.

A verifier is better together with request token, but it is not required. A verifier is used together with request token for exchanging access token, it has an expire time, in this case, it would be a better design if you put them in a cache.

The implemented code looks like:

```
@oauth.verifiersetter
def save_verifier(verifier, token, *args, **kwargs):
    data = Verifier(
        verifier=verifier['oauth_verifier'],
        request_token=token,
        user=get_current_user()
    )
    return data.save()
```

class flask_oauthlib.provider.**OAuth1RequestValidator**(*clientgetter*, *tokengetter*, *tokensetter*, *grantgetter*, *grantsetter*, *noncegetter*, *noncesetter*, *verifiergetter*, *verifiersetter*, *config=None*)

Subclass of Request Validator.

Parameters

- **clientgetter** – a function to get client object
- **tokengetter** – a function to get access token
- **tokensetter** – a function to save access token
- **grantgetter** – a function to get request token
- **grantsetter** – a function to save request token
- **noncegetter** – a function to get nonce and timestamp
- **noncesetter** – a function to save nonce and timestamp

allowed_signature_methods
Allowed signature methods.

Default value: SIGNATURE_HMAC and SIGNATURE_RSA.

You can customize with Flask Config:

- OAUTH1_PROVIDER_SIGNATURE_METHODS

enforce_ssl
Enforce SSL request.

Default is True. You can customize with:

- OAUTH1_PROVIDER_ENFORCE_SSL

get_access_token_secret(*client_key*, *token*, *request*)
Get access token secret.

The access token object should a `secret` attribute.

get_client_secret(*client_key*, *request*)
Get client secret.

The client object must has `client_secret` attribute.

get_default_realms(*client_key*, *request*)
Default realms of the client.

get_realms(*token*, *request*)
Realms for this request token.

get_redirect_uri(*token*, *request*)
Redirect uri for this request token.

get_request_token_secret(*client_key*, *token*, *request*)
Get request token secret.

The request token object should a `secret` attribute.

get_rsa_key(*client_key*, *request*)
Retrieves a previously stored client provided RSA key.

invalidate_request_token(*client_key*, *request_token*, *request*)
Invalidates a used request token.

save_access_token(*token*, *request*)
Save access token to database.

A tokensetter is required, which accepts a token and request parameters:

```
def tokensetter(token, request):
    access_token = Token(
        client=request.client,
        user=request.user,
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        realms=token['oauth_authorized_realms'],
    )
    return access_token.save()
```

save_request_token(*token*, *request*)
Save request token to database.

A grantsetter is required, which accepts a token and request parameters:

```
def grantsetter(token, request):
    grant = Grant(
        token=token['oauth_token'],
        secret=token['oauth_token_secret'],
        client=request.client,
        redirect_uri=oauth.redirect_uri,
        realms=request.realms,
    )
    return grant.save()
```

save_verifier(*token*, *verifier*, *request*)
Save verifier to database.

A verifiersetter is required. It would be better to combine request token and verifier together:

```
def verifiersetter(token, verifier, request):
    tok = Grant.query.filter_by(token=token).first()
    tok.verifier = verifier['oauth_verifier']
    tok.user = get_current_user()
    return tok.save()
```

Note:

A user is required on verifier, remember to attach current user to verifier.

---

validate_access_token(*client_key*, *token*, *request*)
Validates access token is available for client.

validate_client_key(*client_key*, *request*)
Validates that supplied client key.

validate_realms(*client_key*, *token*, *request*, *uri=None*, *realms=None*)
Check if the token has permission on those realms.

validate_redirect_uri(*client_key*, *redirect_uri*, *request*)
Validate if the redirect_uri is allowed by the client.

validate_request_token(*client_key*, *token*, *request*)
Validates request token is available for client.

validate_timestamp_and_nonce(*client_key*, *timestamp*, *nonce*, *request*, *request_token=None*, *access_token=None*)
Validate the timestamp and nonce is used or not.

validate_verifier(*client_key*, *token*, *verifier*, *request*)
Validate verifier exists.

verify_realms(*token*, *realms*, *request*)
Verify if the realms match the requested realms.

verify_request_token(*token*, *request*)
Verify if the request token is existed.

3.1.3 OAuth2 Provider

class flask_oauthlib.provider.**OAuth2Provider**(*app=None*, *validator_class=None*)
Provide secure services using OAuth2.

The server should provide an authorize handler and a token hander, But before the handlers are implemented, the server should provide some getters for the validation.

Like many other Flask extensions, there are two usage modes. One is binding the Flask app instance:

```
app = Flask(__name__)
oauth = OAuth2Provider(app)
```

The second possibility is to bind the Flask app later:

```
oauth = OAuth2Provider()

def create_app():
    app = Flask(__name__)
    oauth.init_app(app)
    return app
```

Configure *tokengetter()* and *tokensetter()* to get and set tokens. Configure *grantgetter()* and *grantsetter()* to get and set grant tokens. Configure *clientgetter()* to get the client.

Configure *usergetter()* if you need password credential authorization.

With everything ready, implement the authorization workflow:

- *authorize_handler()* for consumer to confirm the grant
- *token_handler()* for client to exchange access token

And now you can protect the resource with scopes:

```
@app.route('/api/user')
@oauth.require_oauth('email', 'username')
def user():
    return jsonify(request.oauth.user)
```

after_request(*f*)
Register functions to be invoked after accessing the resource.

The function accepts `valid` and `request` as parameters, and it should return a tuple of them:

```
@oauth.after_request
def valid_after_request(valid, oauth):
    if oauth.user in black_list:
        return False, oauth
    return valid, oauth
```

authorize_handler(*f*)

Authorization handler decorator.

This decorator will sort the parameters and headers out, and pre validate everything:

```
@app.route('/oauth/authorize', methods=['GET', 'POST'])
@oauth.authorize_handler
def authorize(*args, **kwargs):
    if request.method == 'GET':
        # render a page for user to confirm the authorization
        return render_template('oauthorize.html')

    confirm = request.form.get('confirm', 'no')
    return confirm == 'yes'
```

before_request(*f*)

Register functions to be invoked before accessing the resource.

The function accepts nothing as parameters, but you can get information from *Flask.request* object. It is usually useful for setting limitation on the client request:

```
@oauth.before_request
def limit_client_request():
    client_id = request.values.get('client_id')
    if not client_id:
        return
    client = Client.get(client_id)
    if over_limit(client):
        return abort(403)

    track_request(client)
```

clientgetter(*f*)

Register a function as the client getter.

The function accepts one parameter *client_id*, and it returns a client object with at least these information:

- client_id: A random string
- client_secret: A random string
- is_confidential: A bool represents if it is confidential
- redirect_uris: A list of redirect uris
- default_redirect_uri: One of the redirect uris
- default_scopes: Default scopes of the client

The client may contain more information, which is suggested:

- allowed_grant_types: A list of grant types
- allowed_response_types: A list of response types
- validate_scopes: A function to validate scopes

Implement the client getter:

```
@oauth.clientgetter
def get_client(client_id):
    client = get_client_model(client_id)
    # Client is an object
    return client
```

confirm_authorization_request()
When consumer confirm the authorization.

error_uri
The error page URI.

When something turns error, it will redirect to this error page. You can configure the error page URI with Flask config:

```
OAUTH2_PROVIDER_ERROR_URI = '/error'
```

You can also define the error page by a named endpoint:

```
OAUTH2_PROVIDER_ERROR_ENDPOINT = 'oauth.error'
```

grantgetter(*f*)
Register a function as the grant getter.

The function accepts *client_id*, *code* and more:

```
@oauth.grantgetter
def grant(client_id, code):
    return get_grant(client_id, code)
```

It returns a grant object with at least these information:

- delete: A function to delete itself

grantsetter(*f*)
Register a function to save the grant code.

The function accepts *client_id*, *code*, *request* and more:

```
@oauth.grantsetter
def set_grant(client_id, code, request, *args, **kwargs):
    save_grant(client_id, code, request.user, request.scopes)
```

init_app(*app*)
This callback can be used to initialize an application for the oauth provider instance.

invalid_response(*f*)
Register a function for responsing with invalid request.

When an invalid request proceeds to *require_oauth()*, we can handle the request with the registered function. The function accepts one parameter, which is an oauthlib Request object:

```
@oauth.invalid_response
def invalid_require_oauth(req):
    return jsonify(message=req.error_message), 401
```

If no function is registered, it will return with `abort(401)`.

require_oauth(**scopes*)
Protect resource with specified scopes.

revoke_handler(*f*)
Access/refresh token revoke decorator.

Any return value by the decorated function will get discarded as defined in [RFC7009].

You can control the access method with the standard flask routing mechanism, as per [RFC7009] it is recommended to only allow the *POST* method:

```
@app.route('/oauth/revoke', methods=['POST'])
@oauth.revoke_handler
def revoke_token():
    pass
```

server
All in one endpoints. This property is created automaticly if you have implemented all the getters and setters.

However, if you are not satisfied with the getter and setter, you can create a validator with *OAuth2RequestValidator*:

```
class MyValidator(OAuth2RequestValidator):
    def validate_client_id(self, client_id):
        # do something
        return True
```

And assign the validator for the provider:

```
oauth._validator = MyValidator()
```

token_handler(*f*)
Access/refresh token handler decorator.

The decorated function should return an dictionary or None as the extra credentials for creating the token response.

You can control the access method with standard flask route mechanism. If you only allow the *POST* method:

```
@app.route('/oauth/token', methods=['POST'])
@oauth.token_handler
def access_token():
    return None
```

tokengetter(*f*)
Register a function as the token getter.

The function accepts an *access_token* or *refresh_token* parameters, and it returns a token object with at least these information:

- access_token: A string token
- refresh_token: A string token
- client_id: ID of the client
- scopes: A list of scopes
- expires: A *datetime.datetime* object

- user: The user object

The implementation of tokengetter should accepts two parameters, one is access_token the other is refresh_token:

```
@oauth.tokengetter
def bearer_token(access_token=None, refresh_token=None):
    if access_token:
        return get_token(access_token=access_token)
    if refresh_token:
        return get_token(refresh_token=refresh_token)
    return None
```

tokensetter(*f*)

Register a function to save the bearer token.

The setter accepts two parameters at least, one is token, the other is request:

```
@oauth.tokensetter
def set_token(token, request, *args, **kwargs):
    save_token(token, request.client, request.user)
```

The parameter token is a dict, that looks like:

```
{
    u'access_token': u'6JwgO77PApxsFCU8Quz0pnL9s23016',
    u'token_type': u'Bearer',
    u'expires_in': 3600,
    u'scope': u'email address'
}
```

The request is an object, that contains an user object and a client object.

usergetter(*f*)

Register a function as the user getter.

This decorator is only required for **password credential** authorization:

```
@oauth.usergetter
def get_user(username, password, client, request,
             *args, **kwargs):
    # client: current request client
    if not client.has_password_credential_permission:
        return None
    user = User.get_user_by_username(username)
    if not user.validate_password(password):
        return None

    # parameter `request` is an OAuthlib Request object.
    # maybe you will need it somewhere
    return user
```

verify_request(*scopes*)

Verify current request, get the oauth data.

If you can't use the `require_oauth` decorator, you can fetch the data in your request body:

```
def your_handler():
    valid, req = oauth.verify_request(['email'])
```

(continues on next page)

(continued from previous page)

```
    if valid:
        return jsonify(user=req.user)
    return jsonify(status='error')
```

class flask_oauthlib.provider.**OAuth2RequestValidator**(*clientgetter*, *tokengetter*, *grantgetter*, *usergetter=None*, *tokensetter=None*, *grantsetter=None*)

Subclass of Request Validator.

Parameters

- **clientgetter** – a function to get client object
- **tokengetter** – a function to get bearer token
- **tokensetter** – a function to save bearer token
- **grantgetter** – a function to get grant token
- **grantsetter** – a function to save grant token

authenticate_client(*request*, **args*, ***kwargs*)

Authenticate itself in other means.

Other means means is described in Section 3.2.1.

authenticate_client_id(*client_id*, *request*, **args*, ***kwargs*)

Authenticate a non-confidential client.

Parameters

- **client_id** – Client ID of the non-confidential client
- **request** – The Request object passed by oauthlib

client_authentication_required(*request*, **args*, ***kwargs*)

Determine if client authentication is required for current request.

According to the rfc6749, client authentication is required in the following cases:

Resource Owner Password Credentials Grant: see Section 4.3.2. Authorization Code Grant: see Section 4.1.3. Refresh Token Grant: see Section 6.

confirm_redirect_uri(*client_id*, *code*, *redirect_uri*, *client*, **args*, ***kwargs*)

Ensure client is authorized to redirect to the redirect_uri.

This method is used in the authorization code grant flow. It will compare redirect_uri and the one in grant token strictly, you can add a *validate_redirect_uri* function on grant for a customized validation.

confirm_scopes(*refresh_token*, *scopes*, *request*, **args*, ***kwargs*)

Ensures the requested scope matches the scope originally granted by the resource owner. If the scope is omitted it is treated as equal to the scope originally granted by the resource owner.

DEPRECATION NOTE: This method will cease to be used in oauthlib>0.4.2, future versions of `oauthlib` use the validator method `get_original_scopes` to determine the scope of the refreshed token.

get_default_redirect_uri(*client_id*, *request*, **args*, ***kwargs*)

Default redirect_uri for the given client.

get_default_scopes(*client_id*, *request*, **args*, ***kwargs*)

Default scopes for the given client.

get_original_scopes(*refresh_token*, *request*, **args*, ***kwargs*)
Get the list of scopes associated with the refresh token.

This method is used in the refresh token grant flow. We return the scope of the token to be refreshed so it can be applied to the new access token.

invalidate_authorization_code(*client_id*, *code*, *request*, **args*, ***kwargs*)
Invalidate an authorization code after use.

We keep the temporary code in a grant, which has a *delete* function to destroy itself.

revoke_token(*token*, *token_type_hint*, *request*, **args*, ***kwargs*)
Revoke an access or refresh token.

save_authorization_code(*client_id*, *code*, *request*, **args*, ***kwargs*)
Persist the authorization code.

save_bearer_token(*token*, *request*, **args*, ***kwargs*)
Persist the Bearer token.

validate_bearer_token(*token*, *scopes*, *request*)
Validate access token.

Parameters

- **token** – A string of random characters
- **scopes** – A list of scopes
- **request** – The Request object passed by oauthlib

The validation validates:

1. if the token is available
2. if the token has expired
3. if the scopes are available

validate_client_id(*client_id*, *request*, **args*, ***kwargs*)
Ensure client_id belong to a valid and active client.

validate_code(*client_id*, *code*, *client*, *request*, **args*, ***kwargs*)
Ensure the grant code is valid.

validate_grant_type(*client_id*, *grant_type*, *client*, *request*, **args*, ***kwargs*)
Ensure the client is authorized to use the grant type requested.

It will allow any of the four grant types (*authorization_code*, *password*, *client_credentials*, *refresh_token*) by default. Implemented *allowed_grant_types* for client object to authorize the request.

It is suggested that *allowed_grant_types* should contain at least *authorization_code* and *refresh_token*.

validate_redirect_uri(*client_id*, *redirect_uri*, *request*, **args*, ***kwargs*)
Ensure client is authorized to redirect to the redirect_uri.

This method is used in the authorization code grant flow and also in implicit grant flow. It will detect if redirect_uri in client's redirect_uris strictly, you can add a *validate_redirect_uri* function on grant for a customized validation.

validate_refresh_token(*refresh_token*, *client*, *request*, **args*, ***kwargs*)
Ensure the token is valid and belongs to the client

This method is used by the authorization code grant indirectly by issuing refresh tokens, resource owner password credentials grant (also indirectly) and the refresh token grant.

validate_response_type(*client_id*, *response_type*, *client*, *request*, **args*, ***kwargs*)
Ensure client is authorized to use the response type requested.

It will allow any of the two (*code*, *token*) response types by default. Implemented *allowed_response_types* for client object to authorize the request.

validate_scopes(*client_id*, *scopes*, *client*, *request*, **args*, ***kwargs*)
Ensure the client is authorized access to requested scopes.

validate_user(*username*, *password*, *client*, *request*, **args*, ***kwargs*)
Ensure the username and password is valid.

Attach user object on request for later using.

3.1.4 Contrib Reference

Here are APIs provided by contributors.

flask_oauthlib.contrib.oauth2.**bind_sqlalchemy**(*provider*, *session*, *user=None*, *client=None*, *token=None*, *grant=None*, *current_user=None*)
Configures the given `OAuth2Provider` instance with the required getters and setters for persistence with SQLAlchemy.

An example of using all models:

```
oauth = OAuth2Provider(app)

bind_sqlalchemy(oauth, session, user=User, client=Client,
                token=Token, grant=Grant, current_user=current_user)
```

You can omit any model if you wish to register the functions yourself. It is also possible to override the functions by registering them afterwards:

```
oauth = OAuth2Provider(app)

bind_sqlalchemy(oauth, session, user=User, client=Client, token=Token)

@oauth.grantgetter
def get_grant(client_id, code):
    pass

@oauth.grantsetter
def set_grant(client_id, code, request, *args, **kwargs):
    pass

# register tokensetter with oauth but keeping the tokengetter
# registered by `SQLAlchemyBinding`
# You would only do this for the token and grant since user and client
# only have getters
@oauth.tokensetter
def set_token(token, request, *args, **kwargs):
    pass
```

Note that current_user is only required if you're using SQLAlchemy for grant caching. If you're using another caching system with GrantCacheBinding instead, omit current_user.

Parameters

- **provider** – `OAuth2Provider` instance

- **session** – A `Session` object
- **user** – `User` model
- **client** – `Client` model
- **token** – `Token` model
- **grant** – `Grant` model
- **current_user** – function that returns a `User` object

`flask_oauthlib.contrib.oauth2.`**bind_cache_grant**(*app*, *provider*, *current_user*, *config_prefix='OAUTH2'*)

Configures an `OAuth2Provider` instance to use various caching systems to get and set the grant token. This removes the need to register `grantgetter()` and `grantsetter()` yourself.

Parameters

- **app** – Flask application instance
- **provider** – `OAuth2Provider` instance
- **current_user** – function that returns an `User` object
- **config_prefix** – prefix for config

A usage example:

```
oauth = OAuth2Provider(app)
app.config.update({'OAUTH2_CACHE_TYPE': 'redis'})

bind_cache_grant(app, oauth, current_user)
```

You can define which cache system you would like to use by setting the following configuration option:

```
OAUTH2_CACHE_TYPE = 'null' // memcache, simple, redis, filesystem
```

For more information on the supported cache systems please visit: Cache

flask_oauthlib.contrib.apps

The bundle of remote app factories for famous third platforms.

Usage:

```
from flask import Flask
from flask_oauthlib.client import OAuth
from flask_oauthlib.contrib.apps import github

app = Flask(__name__)
oauth = OAuth(app)

github.register_to(oauth, scope=['user:email'])
github.register_to(oauth, name='github2')
```

Of course, it requires consumer keys in your config:

```
GITHUB_CONSUMER_KEY = ''
GITHUB_CONSUMER_SECRET = ''
GITHUB2_CONSUMER_KEY = ''
GITHUB2_CONSUMER_SECRET = ''
```

Some apps with OAuth 1.0a such as Twitter could not accept the `scope` argument.

Contributed by: tonyseek

`flask_oauthlib.contrib.apps.`**`douban`**`()`
The OAuth app for douban.com API.

Parameters **`scope`** – optional. default: `['douban_basic_common']`. see also: http://developers.douban.com/wiki/?title=oauth2

`flask_oauthlib.contrib.apps.`**`dropbox`**`()`
The OAuth app for Dropbox API.

`flask_oauthlib.contrib.apps.`**`facebook`**`()`
The OAuth app for Facebook API.

Parameters **`scope`** – optional. default: `['email']`.

`flask_oauthlib.contrib.apps.`**`github`**`()`
The OAuth app for GitHub API.

Parameters **`scope`** – optional. default: `['user:email']`.

`flask_oauthlib.contrib.apps.`**`google`**`()`
The OAuth app for Google API.

Parameters **`scope`** – optional. default: `['email']`.

`flask_oauthlib.contrib.apps.`**`linkedin`**`()`
The OAuth app for LinkedIn API.

Parameters **`scope`** – optional. default: `['r_basicprofile']`

`flask_oauthlib.contrib.apps.`**`twitter`**`()`
The OAuth app for Twitter API.

`flask_oauthlib.contrib.apps.`**`weibo`**`()`
The OAuth app for weibo.com API.

Parameters **`scope`** – optional. default: `['email']`

CHAPTER 4

Additional Notes

Contribution guide, legal information and changelog are here.

4.1 Contributing

First, please do contribute! There are more than one way to contribute, and I will appreciate any way you choose.

- introduce Flask-OAuthlib to your friends, let Flask-OAuthlib to be known
- discuss Flask-OAuthlib , and submit bugs with github issues
- improve documentation for Flask-OAuthlib
- send patch with github pull request

English and Chinese issues are acceptable, talk in your favorite language.

Pull request and git commit message **must be in English**, if your commit message is in other language, it will be rejected.

4.1.1 Issues

When you submit an issue, please format your content, a readable content helps a lot. You should have a little knowledge on Markdown.

Code talks. If you can't make yourself understood, show me the code. Please make your case as simple as possible.

4.1.2 Codebase

The codebase of Flask-OAuthlib is highly tested and **PEP 8** compatible, as a way to guarantee functionality and keep all code written in a good style.

You should follow the code style. Here are some tips to make things simple:

- When you cloned this repo, run `pip install -r requirements.txt`
- Check the code style with `make lint`
- Check the test with `make test`
- Check the test coverage with `make coverage`

4.1.3 Git Help

Something you should know about git.

- don't add any code on the master branch, create a new one
- don't add too many code in one pull request
- all featured branches should be based on the master branch
- don't merge any code yourself

Take an example, if you want to add feature A and feature B, you should have two branches:

```
$ git branch feature-A
$ git checkout feature-A
```

Now code on feature-A branch, and when you finish feature A:

```
$ git checkout master
$ git branch feature-B
$ git checkout feature-B
```

All branches must be based on the master branch. If your feature-B needs feature-A, you should send feature-A first, and wait for its merging. We may reject feature-A, and you should stop feature-B.

Keep your master branch the same with upstream:

```
$ git remote add upstream git@github.com:lepture/flask-oauthlib.git
$ git pull upstream master
```

And don't change any code on master branch.

4.2 Changelog

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

4.2.1 Version 0.9.5

Released on May 16, 2018

- Fix error handlers
- Update supported OAuthlib
- Add support for string type token

4.2.2 Version 0.9.4

Released on Jun 9, 2017

- Handle HTTP Basic Auth for client's access to token endpoint (#301)
- Allow having access tokens without expiration date (#311)
- Log exception traceback. (#281)

4.2.3 Version 0.9.3

Released on Jun 2, 2016

- Revert the wrong implement of non credential oauth2 require auth
- Catch all exceptions in OAuth2 providers
- Bugfix for examples, docs and other things

4.2.4 Version 0.9.2

Released on Nov 3, 2015

- Bugfix in client parse_response when body is none.
- Update contrib client by @tonyseek
- Typo fix for OAuth1 provider
- Fix OAuth2 provider on non credential clients by @Fleurer

4.2.5 Version 0.9.1

Released on Mar 9, 2015

- Improve on security.
- Fix on contrib client.

4.2.6 Version 0.9.0

Released on Feb 3, 2015

- New feature for contrib client, which will become the official client in the future via #136 and #176.
- Add appropriate headers when making POST request for access toke via #169.
- Use a local copy of instance 'request_token_params' attribute to avoid side effects via #177.
- Some minor fixes of contrib by Hsiaoming Yang.

4.2.7 Version 0.8.0

Released on Dec 3, 2014

- New feature for generating refresh tokens
- Add new function `OAuth2Provider.verify_request()` for non vanilla Flask projects
- Some small bugfixes

4.2.8 Version 0.7.0

Released on Aug 20, 2014

- Deprecated *OAuthRemoteApp.authorized_handler()* in favor of *OAuthRemoteApp.authorized_response()*.
- Add revocation endpoint via #131.
- Handle unknown exceptions in providers.
- Add PATCH method for client via #134.

4.2.9 Version 0.6.0

Released on Jul 29, 2014

- Compatible with OAuthLib 0.6.2 and 0.6.3
- Add invalid_response decorator to handle invalid request
- Add error_message for OAuthLib Request.

4.2.10 Version 0.5.0

Released on May 13, 2014

- Add `contrib.apps` module, thanks for tonyseek via #94.
- Status code changed to 401 for invalid access token via #93.
- **Security bug** for access token via #92.
- Fix for client part, request token params for OAuth1 via #91.
- **API change** for `oauth.require_oauth` via #89.
- Fix for OAuth2 provider, support client authentication for authorization-code grant type via #86.
- Fix client_credentials logic in validate_grant_type via #85.
- Fix for client part, pass access token method via #83.
- Fix for OAuth2 provider related to confidential client via #82.

Upgrade From 0.4.x to 0.5.0

API for OAuth providers `oauth.require_oauth` has changed.

Before the change, you would write code like:

```
@app.route('/api/user')
@oauth.require_oauth('email')
def user(req):
    return jsonify(req.user)
```

After the change, you would write code like:

```
from flask import request

@app.route('/api/user')
@oauth.require_oauth('email')
def user():
    return jsonify(request.oauth.user)
```

Thanks Stian Prestholdt and Jiangge Zhang.

4.2.11 Version 0.4.3

Released on Feb 18, 2014

- OAuthlib released 0.6.1, which caused a bug in oauth2 provider.
- Validation for scopes on oauth2 right via #72.
- Handle empty response for application/json via #69.

4.2.12 Version 0.4.2

Released on Jan 3, 2014

Happy New Year!

- Add param `state` in authorize method via #63.
- Bugfix for encoding error in Python 3 via #65.

4.2.13 Version 0.4.1

Released on Nov 25, 2013

- Add access_token on request object via #53.
- Bugfix for lazy loading configuration via #55.

4.2.14 Version 0.4.0

Released on Nov 12, 2013

- Redesign contrib library.
- A new way for lazy loading configuration via #51.

- Some bugfixes.

4.2.15 Version 0.3.4

Released on Oct 31, 2013

- Bugfix for client missing a string placeholder via #49.
- Bugfix for client property getter via #48.

4.2.16 Version 0.3.3

Released on Oct 4, 2013

- Support for token generator in OAuth2 Provider via #42.
- Improve client part, improve test cases.
- Fix scope via #44.

4.2.17 Version 0.3.2

Released on Sep 13, 2013

- Upgrade oauthlib to 0.6
- A quick bugfix for request token params via #40.

4.2.18 Version 0.3.1

Released on Aug 22, 2013

- Add contrib module via #15. We are still working on it, take your own risk.
- Add example of linkedin via #35.
- Compatible with new proposals of oauthlib.
- Bugfix for client part.
- Backward compatible for lower version of Flask via #37.

4.2.19 Version 0.3.0

Released on July 10, 2013.

- OAuth1 Provider available. Documentation at *OAuth1 Server*. :)
- Add `before_request` and `after_request` via #22.
- Lazy load configuration for client via #23. Documentation at *Lazy Configuration*.
- Python 3 compatible now.

4.2.20 Version 0.2.0

Released on June 19, 2013.

- OAuth2 Provider available. Documentation at *OAuth2 Server*. :)
- Make client part testable.
- Change extension name of client from `oauth-client` to `oauthlib.client`.

4.2.21 Version 0.1.1

Released on May 23, 2013.

- Fix setup.py

4.2.22 Version 0.1.0

First public preview release on May 18, 2013.

4.3 Authors

Flask-OAuthlib is written and maintained by Hsiaoming Yang <me@lepture.com>.

4.3.1 Contributors

People who send patches and suggestions:

- Hsiaoming Yang http://github.com/lepture
- Randy Topliffe https://github.com/Taar
- Mackenzie Blake Thompson https://github.com/flippmoke
- Ib Lundgren https://github.com/ib-lundgren
- Jiangge Zhang https://github.com/tonyseek
- Stian Prestholdt https://github.com/stianpr

Find more contributors on Github.

Python Module Index

f

Index

H

I

L

N

O

P

R

S

T

U

V

W