Async-OAuthlib

Release 0.0.7

Nov 06, 2020

Contents

Requests-OAuthlib uses the Python Aiohttp and OAuthlib libraries to provide an easy-to-use Python interface for building OAuth1 and OAuth2 clients.

CHAPTER 1

Overview

A simple Flask application which connects to the Github OAuth2 API looks approximately like this:

```
from async_oauthlib import OAuth2Session

from quart import Quart, request, redirect, session, url_for
from quart.json import jsonify

# This information is obtained upon registration of a new GitHub
client_id = "<your client key>"
client_secret = "<your client secret>"
authorization_base_url = 'https://github.com/login/oauth/authorize'
token_url = 'https://github.com/login/oauth/access_token'

@app.route("/login")
async def login():
    github = OAuth2Session(client_id)
    authorization_url, state = github.authorization_url(authorization_base_url)

    # State is used to prevent CSRF, keep this for later.
    session['oauth_state'] = state
    return redirect(authorization_url)

@app.route("/callback")
async def callback():
    github = OAuth2Session(client_id, state=session['oauth_state'])
    token = await github.fetch_token(token_url, client_secret=client_secret,␣
↪authorization_response=request.url)

    return jsonify((await github.get('https://api.github.com/user')).json())
```

The above is a truncated example. A full working example is available here: *Web App Example of OAuth 2 web application flow*

CHAPTER 2

Installation

Requests-OAuthlib can be installed with pip:

```
$ pip install Async-OAuthlib
```

CHAPTER 3

Getting Started:

3.1 OAuth 1 Workflow

You will be forced to go through a few steps when you are using OAuth. Below is an example of the most common OAuth workflow using HMAC-SHA1 signed requests where the signature is supplied in the Authorization header.

The example assumes an interactive prompt which is good for demonstration but in practice you will likely be using a web application (which makes authorizing much less awkward since you can simply redirect).

The guide will show two ways of carrying out the OAuth1 workflow. One using the authentication helper OAuth1 and the alternative using OAuth1Session. The latter is usually more convenient and requires less code.

3.1.1 Workflow example showing use of both OAuth1 and OAuth1Session

0. Manual client signup with the OAuth provider (i.e. Google, Twitter) to get a set of client credentials. Usually a client key and secret. Client might sometimes be referred to as consumer. For example:

```
>>> # Using OAuth1Session
>>> from requests_oauthlib import OAuth1Session

>>> # Using OAuth1 auth helper
>>> import requests
>>> from requests_oauthlib import OAuth1

>>> client_key = '...'
>>> client_secret = '...'
```

1. Obtain a request token which will identify you (the client) in the next step. At this stage you will only need your client key and secret.

```
>>> request_token_url = 'https://api.twitter.com/oauth/request_token'

>>> # Using OAuth1Session
```

(continues on next page)

(continued from previous page)

```
>>> oauth = OAuth1Session(client_key, client_secret=client_secret)
>>> fetch_response = oauth.fetch_request_token(request_token_url)
{
    "oauth_token": "Z6eEdO8MOmk394WozF5oKyuAv855l4Mlqo7hhlSLik",
    "oauth_token_secret": "Kd75W4OQfb2oJTV0vzGzeXftVAwgMnEK9MumzYcM"
}
>>> resource_owner_key = fetch_response.get('oauth_token')
>>> resource_owner_secret = fetch_response.get('oauth_token_secret')

>>> # Using OAuth1 auth helper
>>> oauth = OAuth1(client_key, client_secret=client_secret)
>>> r = requests.post(url=request_token_url, auth=oauth)
>>> r.content
"oauth_token=Z6eEdO8MOmk394WozF5oKyuAv855l4Mlqo7hhlSLik&oauth_token_
↪secret=Kd75W4OQfb2oJTV0vzGzeXftVAwgMnEK9MumzYcM"
>>> from urlparse import parse_qs
>>> credentials = parse_qs(r.content)
>>> resource_owner_key = credentials.get('oauth_token')[0]
>>> resource_owner_secret = credentials.get('oauth_token_secret')[0]
```

2. Obtain authorization from the user (resource owner) to access their protected resources (images, tweets, etc.). This is commonly done by redirecting the user to a specific url to which you add the request token as a query parameter. Note that not all services will give you a verifier even if they should. Also the oauth_token given here will be the same as the one in the previous step.

```
>>> base_authorization_url = 'https://api.twitter.com/oauth/authorize'

>>> # Using OAuth1Session
>>> authorization_url = oauth.authorization_url(base_authorization_url)
>>> print 'Please go here and authorize,', authorization_url
>>> redirect_response = raw_input('Paste the full redirect URL here: ')
>>> oauth_response = oauth.parse_authorization_response(redirect_response)
{
    "oauth_token": "Z6eEdO8MOmk394WozF5oKyuAv855l4Mlqo7hhlSLik",
    "oauth_verifier": "sdflk3450FASDLJasd2349dfs"
}
>>> verifier = oauth_response.get('oauth_verifier')

>>> # Using OAuth1 auth helper
>>> authorize_url = base_authorization_url + '?oauth_token='
>>> authorize_url = authorize_url + resource_owner_key
>>> print 'Please go here and authorize,', authorize_url
>>> verifier = raw_input('Please input the verifier')
```

3. Obtain an access token from the OAuth provider. Save this token as it can be re-used later. In this step we will re-use most of the credentials obtained uptil this point.

```
>>> access_token_url = 'https://api.twitter.com/oauth/access_token'

>>> # Using OAuth1Session
>>> oauth = OAuth1Session(client_key,
                          client_secret=client_secret,
                          resource_owner_key=resource_owner_key,
                          resource_owner_secret=resource_owner_secret,
                          verifier=verifier)
>>> oauth_tokens = oauth.fetch_access_token(access_token_url)
```

(continues on next page)

(continued from previous page)

```
{
    "oauth_token": "6253282-eWudHldSbIaelX7swmsiHImEL4KinwaGloHANdrY",
    "oauth_token_secret": "2EEfA6BG3ly3sR3RjE0IBSnlQu4ZrUzPiYKmrkVU"
}
>>> resource_owner_key = oauth_tokens.get('oauth_token')
>>> resource_owner_secret = oauth_tokens.get('oauth_token_secret')

>>> # Using OAuth1 auth helper
>>> oauth = OAuth1(client_key,
                   client_secret=client_secret,
                   resource_owner_key=resource_owner_key,
                   resource_owner_secret=resource_owner_secret,
                   verifier=verifier)
>>> r = requests.post(url=access_token_url, auth=oauth)
>>> r.content
"oauth_token=6253282-eWudHldSbIaelX7swmsiHImEL4KinwaGloHANdrY&oauth_token_
↪secret=2EEfA6BG3ly3sR3RjE0IBSnlQu4ZrUzPiYKmrkVU"
>>> credentials = parse_qs(r.content)
>>> resource_owner_key = credentials.get('oauth_token')[0]
>>> resource_owner_secret = credentials.get('oauth_token_secret')[0]
```

4. Access protected resources. OAuth1 access tokens typically do not expire and may be re-used until revoked by the user or yourself.

```
>>> protected_url = 'https://api.twitter.com/1/account/settings.json'

>>> # Using OAuth1Session
>>> oauth = OAuth1Session(client_key,
                          client_secret=client_secret,
                          resource_owner_key=resource_owner_key,
                          resource_owner_secret=resource_owner_secret)
>>> r = oauth.get(protected_url)

>>> # Using OAuth1 auth helper
>>> oauth = OAuth1(client_key,
                   client_secret=client_secret,
                   resource_owner_key=resource_owner_key,
                   resource_owner_secret=resource_owner_secret)
>>> r = requests.get(url=protected_url, auth=oauth)
```

3.1.2 Signature placement - header, query or body?

OAuth takes many forms, so let's take a look at a few different forms:

```
import requests
from requests_oauthlib import OAuth1

url = u'https://api.twitter.com/1/account/settings.json'

client_key = u'...'
client_secret = u'...'
resource_owner_key = u'...'
resource_owner_secret = u'...'
```

Header signing (recommended):

```
headeroauth = OAuth1(client_key, client_secret,
                     resource_owner_key, resource_owner_secret,
                     signature_type='auth_header')
r = requests.get(url, auth=headeroauth)
```

Query signing:

```
queryoauth = OAuth1(client_key, client_secret,
                    resource_owner_key, resource_owner_secret,
                    signature_type='query')
r = requests.get(url, auth=queryoauth)
```

Body signing:

```
bodyoauth = OAuth1(client_key, client_secret,
                   resource_owner_key, resource_owner_secret,
                   signature_type='body')

r = requests.post(url, auth=bodyoauth)
```

3.1.3 Signature types - HMAC (most common), RSA, Plaintext

OAuth1 defaults to using HMAC and examples can be found in the previous sections.

Plaintext work on the same credentials as HMAC and the only change you will need to make when using it is to add signature_type='PLAINTEXT' to the OAuth1 constructor:

```
headeroauth = OAuth1(client_key, client_secret,
                     resource_owner_key, resource_owner_secret,
                     signature_method='PLAINTEXT')
```

RSA is different in that it does not use client_secret nor resource_owner_secret. Instead it uses public and private keys. The public key is provided to the OAuth provider during client registration. The private key is used to sign requests. The previous section can be summarized as:

```
key = open("your_rsa_key.pem").read()

queryoauth = OAuth1(client_key, signature_method=SIGNATURE_RSA,
                    rsa_key=key, signature_type='query')
headeroauth = OAuth1(client_key, signature_method=SIGNATURE_RSA,
                     rsa_key=key, signature_type='auth_header')
bodyoauth = OAuth1(client_key, signature_method=SIGNATURE_RSA,
                   rsa_key=key, signature_type='body')
```

3.2 OAuth 2 Workflow

3.2.1 Introduction

The following sections provide some example code that demonstrates some of the possible OAuth2 flows you can use with requests-oauthlib. We provide four examples: one for each of the grant types defined by the OAuth2 RFC. These grant types (or workflows) are the Authorization Code Grant (or Web Application Flow), the Implicit Grant (or Mobile Application Flow), the Resource Owner Password Credentials Grant (or, more succinctly, the Legacy Application Flow), and the Client Credentials Grant (or Backend Application Flow).

Available Workflows

There are four core work flows:

1. *Authorization Code Grant* (Web Application Flow).
2. *Implicit Grant* (Mobile Application flow).
3. *Resource Owner Password Credentials Grant* (Legacy Application flow).
4. *Client Credentials Grant* (Backend Application flow).

3.2.2 Web Application Flow

The steps below outline how to use the default Authorization Grant Type flow to obtain an access token and fetch a protected resource. In this example the provider is Google and the protected resource is the user's profile.

0. Obtain credentials from your OAuth provider manually. At minimum you will need a `client_id` but likely also a `client_secret`. During this process you might also be required to register a default redirect URI to be used by your application. Save these things in your Python script:

```
>>> client_id = r'your_client_id'
>>> client_secret = r'your_client_secret'
>>> redirect_uri = 'https://your.callback/uri'
```

1. User authorization through redirection. First we will create an authorization url from the base URL given by the provider and the credentials previously obtained. In addition most providers will request that you ask for access to a certain scope. In this example we will ask Google for access to the email address of the user and the users profile.

```
# Note that these are Google specific scopes
>>> scope = ['https://www.googleapis.com/auth/userinfo.email',
             'https://www.googleapis.com/auth/userinfo.profile']
>>> oauth = OAuth2Session(client_id, redirect_uri=redirect_uri,
                          scope=scope)
>>> authorization_url, state = oauth.authorization_url(
        'https://accounts.google.com/o/oauth2/auth',
        # access_type and prompt are Google specific extra
        # parameters.
        access_type="offline", prompt="select_account")

>>> print 'Please go to %s and authorize access.' % authorization_url
>>> authorization_response = raw_input('Enter the full callback URL')
```

2. Fetch an access token from the provider using the authorization code obtained during user authorization.

```
>>> token = await oauth.fetch_token(
        'https://accounts.google.com/o/oauth2/token',
        authorization_response=authorization_response,
        # Google specific extra parameter used for client
        # authentication
        client_secret=client_secret)
```

3. Access protected resources using the access token you just obtained. For example, get the users profile info.

```
>>> r = await oauth.get('https://www.googleapis.com/oauth2/v1/userinfo')
>>> # Enjoy =)
```

3.2.3 Mobile Application Flow

The steps below outline how to use the Implicit Code Grant Type flow to obtain an access token.

0. You will need the following settings.

```
>>> client_id = 'your_client_id'
>>> scopes = ['scope_1', 'scope_2']
>>> auth_url = 'https://your.oauth2/auth'
```

1. Get the authorization_url

```
>>> from oauthlib.oauth2 import MobileApplicationClient
>>> from requests_oauthlib import OAuth2Session
>>> oauth = OAuth2Session(client=MobileApplicationClient(client_id=client_id),␣
↪scope=scopes)
>>> authorization_url, state = oauth.authorization_url(auth_url)
```

2. Fetch an access token from the provider.

```
>>> response = await oauth.get(authorization_url)
>>> oauth.token_from_fragment(response.url)
```

3.2.4 Legacy Application Flow

The steps below outline how to use the Resource Owner Password Credentials Grant Type flow to obtain an access token.

0. You will need the following settings. `client_secret` is optional depending on the provider.

```
>>> client_id = 'your_client_id'
>>> client_secret = 'your_client_secret'
>>> username = 'your_username'
>>> password = 'your_password'
```

1. Fetch an access token from the provider.

```
>>> from oauthlib.oauth2 import LegacyApplicationClient
>>> from requests_oauthlib import OAuth2Session
>>> oauth = OAuth2Session(client=LegacyApplicationClient(client_id=client_id))
>>> token = await oauth.fetch_token(token_url='https://somesite.com/oauth2/token',
        username=username, password=password, client_id=client_id,
        client_secret=client_secret)
```

3.2.5 Backend Application Flow

The steps below outline how to use the Resource Owner Client Credentials Grant Type flow to obtain an access token.

0. Obtain credentials from your OAuth provider. At minimum you will need a `client_id` and `client_secret`.

```
>>> client_id = 'your_client_id'
>>> client_secret = 'your_client_secret'
```

1. Fetch an access token from the provider.

```
>>> from oauthlib.oauth2 import BackendApplicationClient
>>> from requests_oauthlib import OAuth2Session
>>> client = BackendApplicationClient(client_id=client_id)
>>> oauth = OAuth2Session(client=client)
>>> token = await oauth.fetch_token(token_url='https://provider.com/
↪oauth2/token', client_id=client_id,
        client_secret=client_secret)
```

If your provider requires that you pass auth credentials in a Basic Auth header, you can do this instead:

```
>>> from oauthlib.oauth2 import BackendApplicationClient
>>> from requests_oauthlib import OAuth2Session
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth(client_id, client_secret)
>>> client = BackendApplicationClient(client_id=client_id)
>>> oauth = OAuth2Session(client=client)
>>> token = await oauth.fetch_token(token_url='https://provider.com/
↪oauth2/token', auth=auth)
```

3.2.6 Refreshing tokens

Certain providers will give you a `refresh_token` along with the `access_token`. These can be used to directly fetch new access tokens without going through the normal OAuth workflow. `requests-oauthlib` provides three methods of obtaining refresh tokens. All of these are dependant on you specifying an accurate `expires_in` in the token.

`expires_in` is a credential given with the access and refresh token indicating in how many seconds from now the access token expires. Commonly, access tokens expire after an hour and the `expires_in` would be `3600`. Without

this it is impossible for `requests-oauthlib` to know when a token is expired as the status code of a request failing due to token expiration is not defined.

If you are not interested in token refreshing, always pass in a positive value for `expires_in` or omit it entirely.

(ALL) Define the token, token saver and needed credentials

```
>>> token = {
...     'access_token': 'eswfld123kjhn1v5423',
...     'refresh_token': 'asdfkljh23490sdf',
...     'token_type': 'Bearer',
...     'expires_in': '-30',     # initially 3600, need to be updated by you
...  }
>>> client_id = r'foo'
>>> refresh_url = 'https://provider.com/token'
>>> protected_url = 'https://provider.com/secret'

>>> # most providers will ask you for extra credentials to be passed along
>>> # when refreshing tokens, usually for authentication purposes.
>>> extra = {
...     'client_id': client_id,
...     'client_secret': r'potato',
... }

>>> # After updating the token you will most likely want to save it.
>>> async def token_saver(token):
...     # save token in database / session
```

(First) Define Try-Catch TokenExpiredError on each request

This is the most basic version in which an error is raised when refresh is necessary but refreshing is done manually.

```
>>> from requests_oauthlib import OAuth2Session
>>> from oauthlib.oauth2 import TokenExpiredError
>>> try:
...     client = OAuth2Session(client_id, token=token)
...     r = await client.get(protected_url)
>>> except TokenExpiredError as e:
...     token = await client.refresh_token(refresh_url, **extra)
...     await token_saver(token)
>>> client = OAuth2Session(client_id, token=token)
>>> r = await client.get(protected_url)
```

(Second) Define automatic token refresh automatic but update manually

This is the, arguably awkward, middle between the basic and convenient refresh methods in which a token is automatically refreshed, but saving the new token is done manually.

```
>>> from requests_oauthlib import OAuth2Session, TokenUpdated
>>> try:
...     client = OAuth2Session(client_id, token=token,
...             auto_refresh_kwargs=extra, auto_refresh_url=refresh_url)
...     r = await client.get(protected_url)
```

(continues on next page)

(continued from previous page)

```
>>> except TokenUpdated as e:
...     await token_saver(e.token)
```

(Third, Recommended) Define automatic token refresh and update

The third and recommended method will automatically fetch refresh tokens and save them. It requires no exception catching and results in clean code. Remember however that you still need to update `expires_in` to trigger the refresh.

```
>>> from requests_oauthlib import OAuth2Session
>>> client = OAuth2Session(client_id, token=token, auto_refresh_url=refresh_url,
...     auto_refresh_kwargs=extra, token_updater=token_saver)
>>> r = await client.get(protected_url)
```

3.2.7 TLS Client Authentication

To use TLS Client Authentication (draft-ietf-oauth-mtls) via a self-signed or CA-issued certificate, pass the certificate in the token request and ensure that the client id is sent in the request:

```
>>> await oauth.fetch_token(token_url='https://somesite.com/oauth2/token',
...     include_client_id=True, cert=('test-client.pem', 'test-client-key.pem'))
```

3.3 Examples

3.3.1 Bitbucket OAuth 1 Tutorial

Start with setting up a new consumer by following the instructions on Bitbucket. When you have obtained a `key` and a `secret` you can try out the command line interactive example below.

```
# Credentials you get from adding a new consumer in bitbucket -> manage account
# -> integrated applications.
>>> key = '<the key you get from bitbucket>'
>>> secret = '<the secret you get from bitbucket>'

>>> # OAuth endpoints given in the Bitbucket API documentation
>>> request_token_url = 'https://bitbucket.org/!api/1.0/oauth/request_token'
>>> authorization_base_url = 'https://bitbucket.org/!api/1.0/oauth/authenticate'
>>> access_token_url = 'https://bitbucket.org/!api/1.0/oauth/access_token'

>>> # 2. Fetch a request token
>>> from requests_oauthlib import OAuth1Session
>>> bitbucket = OAuth1Session(key, client_secret=secret,
>>>         callback_uri='http://127.0.0.1/cb')
>>> bitbucket.fetch_request_token(request_token_url)

>>> # 3. Redirect user to Bitbucket for authorization
>>> authorization_url = bitbucket.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # 4. Get the authorization verifier code from the callback url
```

(continues on next page)

(continued from previous page)

```
>>> redirect_response = raw_input('Paste the full redirect URL here:')
>>> bitbucket.parse_authorization_response(redirect_response)

>>> # 5. Fetch the access token
>>> bitbucket.fetch_access_token(access_token_url)

>>> # 6. Fetch a protected resource, i.e. user profile
>>> r = bitbucket.get('https://bitbucket.org/api/1.0/user')
>>> print r.content
```

3.3.2 GitHub OAuth 2 Tutorial

Setup credentials following the instructions on GitHub. When you have obtained a `client_id` and a `client_secret` you can try out the command line interactive example below.

```
>>> # Credentials you get from registering a new application
>>> client_id = '<the id you get from github>'
>>> client_secret = '<the secret you get from github>'

>>> # OAuth endpoints given in the GitHub API documentation
>>> authorization_base_url = 'https://github.com/login/oauth/authorize'
>>> token_url = 'https://github.com/login/oauth/access_token'

>>> from requests_oauthlib import OAuth2Session
>>> github = OAuth2Session(client_id)

>>> # Redirect user to GitHub for authorization
>>> authorization_url, state = github.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = raw_input('Paste the full redirect URL here:')

>>> # Fetch the access token
>>> github.fetch_token(token_url, client_secret=client_secret,
>>>         authorization_response=redirect_response)

>>> # Fetch a protected resource, i.e. user profile
>>> r = github.get('https://api.github.com/user')
>>> print r.content
```

3.3.3 Google OAuth 2 Tutorial

Setup a new web project in the Google Cloud Console When you have obtained a `client_id`, `client_secret` and registered a callback URL then you can try out the command line interactive example below.

```
>>> # Credentials you get from registering a new application
>>> client_id = '<the id you get from google>.apps.googleusercontent.com'
>>> client_secret = '<the secret you get from google>'
>>> redirect_uri = 'https://your.registered/callback'

>>> # OAuth endpoints given in the Google API documentation
>>> authorization_base_url = "https://accounts.google.com/o/oauth2/v2/auth"
```

(continues on next page)

(continued from previous page)

```
>>> token_url = "https://www.googleapis.com/oauth2/v4/token"
>>> scope = [
...     "https://www.googleapis.com/auth/userinfo.email",
...     "https://www.googleapis.com/auth/userinfo.profile"
... ]

>>> from requests_oauthlib import OAuth2Session
>>> google = OAuth2Session(client_id, scope=scope, redirect_uri=redirect_uri)

>>> # Redirect user to Google for authorization
>>> authorization_url, state = google.authorization_url(authorization_base_url,
...     # offline for refresh token
...     # force to always make user click authorize
...     access_type="offline", prompt="select_account")
>>> print('Please go here and authorize:', authorization_url)

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = input('Paste the full redirect URL here: ')

>>> # Fetch the access token
>>> google.fetch_token(token_url, client_secret=client_secret,
...         authorization_response=redirect_response)

>>> # Fetch a protected resource, i.e. user profile
>>> r = google.get('https://www.googleapis.com/oauth2/v1/userinfo')
>>> print(r.content)
```

3.3.4 Facebook OAuth 2 Tutorial

Setup a new web application client in the Facebook APP console When you have obtained a `client_id`, `client_secret` and registered a callback URL then you can try out the command line interactive example below.

```
>>> # Credentials you get from registering a new application
>>> client_id = '<the id you get from facebook>'
>>> client_secret = '<the secret you get from facebook>'

>>> # OAuth endpoints given in the Facebook API documentation
>>> authorization_base_url = 'https://www.facebook.com/dialog/oauth'
>>> token_url = 'https://graph.facebook.com/oauth/access_token'
>>> redirect_uri = 'https://localhost/'     # Should match Site URL

>>> from requests_oauthlib import OAuth2Session
>>> from requests_oauthlib.compliance_fixes import facebook_compliance_fix
>>> facebook = OAuth2Session(client_id, redirect_uri=redirect_uri)
>>> facebook = facebook_compliance_fix(facebook)

>>> # Redirect user to Facebook for authorization
>>> authorization_url, state = facebook.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = raw_input('Paste the full redirect URL here:')

>>> # Fetch the access token
```

(continues on next page)

(continued from previous page)

```
>>> facebook.fetch_token(token_url, client_secret=client_secret,
..>                       authorization_response=redirect_response)

>>> # Fetch a protected resource, i.e. user profile
>>> r = facebook.get('https://graph.facebook.com/me?')
>>> print r.content
```

3.3.5 Fitbit OAuth 2 (Mobile Application Flow) Tutorial

This makes use of the Implicit Grant Flow to obtain an access token for the Fitbit API. Register a new client application there with a callback URL, and have your client ID handy. Based on an another example of the Mobile Application Flow.

```
>>> import requests
>>> from requests_oauthlib import OAuth2Session
>>> from oauthlib.oauth2 import MobileApplicationClient

# Set up your client ID and scope: the scope must match that which you requested when␣
↪you set up your application.
>>> client_id = "<your client ID here>"
>>> scope = ["activity", "heartrate", "location", "nutrition", "profile", "settings",
↪"sleep", "social", "weight"]

# Initialize client
>>> client = MobileApplicationClient(client_id)
>>> fitbit = OAuth2Session(client_id, client=client, scope=scope)
>>> authorization_url = "https://www.fitbit.com/oauth2/authorize"

# Grab the URL for Fitbit's authorization page.
>>> auth_url, state = fitbit.authorization_url(authorization_url)
>>> print("Visit this page in your browser: {}".format(auth_url))

# After authenticating, Fitbit will redirect you to the URL you specified in your␣
↪application settings. It contains the access token.
>>> callback_url = input("Paste URL you get back here: ")

# Now we extract the token from the URL to make use of it.
>>> fitbit.token_from_fragment(callback_url)

# We can also store the token for use later.
>>> token = fitbit['token']

# At this point, assuming nothing blew up, we can make calls to the API as normal,␣
↪for example:
>>> r = fitbit.get('https://api.fitbit.com/1/user/-/sleep/goal.json')
```

3.3.6 LinkedIn OAuth 2 Tutorial

Setup credentials following the instructions on LinkedIn. When you have obtained a `client_id` and a `client_secret` you can try out the command line interactive example below.

```
>>> # Imports
>>> import os
```

(continues on next page)

(continued from previous page)

```
>>> from requests_oauthlib import OAuth2Session

>>> # Set environment variables
>>> os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = '1'

>>> # Credentials you get from registering a new application
>>> client_id = '<the id you get from linkedin>'
>>> client_secret = '<the secret you get from linkedin>'

>>> # LinkedIn OAuth2 requests require scope and redirect_url parameters.
>>> # Ensure these values match the auth values in your LinkedIn App
>>> # (see auth tab on LinkedIn Developer page)
>>> scope = ['r_liteprofile']
>>> redirect_url = 'http://127.0.0.1'

>>> # OAuth endpoints given in the LinkedIn API documentation
>>> authorization_base_url = 'https://www.linkedin.com/oauth/v2/authorization'
>>> token_url = 'https://www.linkedin.com/oauth/v2/accessToken'

>>> linkedin = OAuth2Session(client_id, redirect_uri='http://127.0.0.1', scope=scope)

>>> # Redirect user to LinkedIn for authorization
>>> authorization_url, state = linkedin.authorization_url(authorization_base_url)
>>> print(f"Please go here and authorize: {authorization_url}")

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = input('Paste the full redirect URL here:')

>>> # Fetch the access token
>>> linkedin.fetch_token(token_url, client_secret=client_secret,
...                      include_client_id=True,
...                      authorization_response=redirect_response)

>>> # Fetch a protected resource, i.e. user profile
>>> r = linkedin.get('https://api.linkedin.com/v2/me')
>>> print(r.content)
```

3.3.7 Outlook Calendar OAuth 2 Tutorial

Create a new web application client in the **'Microsoft Application Registration Portal'_** When you have obtained a `client_id`, `client_secret` and registered a callback URL then you can try out the command line interactive example below.

```
>>> # This information is obtained upon registration of a new Outlook Application
>>> client_id = '<the id you get from outlook>'
>>> client_secret = '<the secret you get from outlook>'

>>> # OAuth endpoints given in Outlook API documentation
>>> authorization_base_url = 'https://login.microsoftonline.com/common/oauth2/v2.0/
↪authorize'
>>> token_url = 'https://login.microsoftonline.com/common/oauth2/v2.0/token'
>>> scope = ['https://outlook.office.com/calendars.readwrite']
>>> redirect_uri = 'https://localhost/'     # Should match Site URL

>>> from requests_oauthlib import OAuth2Session
```

(continues on next page)

(continued from previous page)

```
>>> outlook = OAuth2Session(client_id,scope=scope,redirect_uri=redirect_uri)

>>> # Redirect  the user owner to the OAuth provider (i.e. Outlook) using an URL with␣
↪a few key OAuth parameters.
>>> authorization_url, state = outlook.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # Get the authorization verifier code from the callback url
>>> redirect_response = raw_input('Paste the full redirect URL here:')

>>> # Fetch the access token
>>> token = outlook.fetch_token(token_url,client_secret=client_secret,authorization_
↪response=redirect_response)

>>> # Fetch a protected resource, i.e. calendar information
>>> o = outlook.get('https://outlook.office.com/api/v1.0/me/calendars')
>>> print o.content
```

3.3.8 Tumblr OAuth1 Tutorial

Register a new application on the tumblr application page. Enter a call back url (can just be http://www.tumblr.com/dashboard) and get the `OAuth Consumer Key` and `Secret Key`.

```
>>> # Credentials from the application page
>>> key = '<the app key>'
>>> secret = '<the app secret>'

>>> # OAuth URLs given on the application page
>>> request_token_url = 'http://www.tumblr.com/oauth/request_token'
>>> authorization_base_url = 'http://www.tumblr.com/oauth/authorize'
>>> access_token_url = 'http://www.tumblr.com/oauth/access_token'

>>> # Fetch a request token
>>> from requests_oauthlib import OAuth1Session
>>> tumblr = OAuth1Session(key, client_secret=secret, callback_uri='http://www.tumblr.
↪com/dashboard')
>>> tumblr.fetch_request_token(request_token_url)

>>> # Link user to authorization page
>>> authorization_url = tumblr.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

>>> # Get the verifier code from the URL
>>> redirect_response = raw_input('Paste the full redirect URL here: ')
>>> tumblr.parse_authorization_response(redirect_response)

>>> # Fetch the access token
>>> tumblr.fetch_access_token(access_token_url)

>>> # Fetch a protected resource
>>> print tumblr.get('http://api.tumblr.com/v2/user/dashboard')
```

3.3.9 Web App Example of OAuth 2 web application flow

OAuth is commonly used by web applications. The example below shows what such a web application might look like using the Flask web framework and GitHub as a provider. It should be easily transferrable to any web framework.

Note: While the flow remains the same across most providers, Github is special in that the `redirect_uri` parameter is optional. This means that it may be necessary to explicitly pass a redirect_uri to the `OAuth2Session` object (e.g. when creating a custom OAuthProvider with `flask-oauthlib`).

```
from requests_oauthlib import OAuth2Session
from flask import Flask, request, redirect, session, url_for
from flask.json import jsonify
import os

app = Flask(__name__)


# This information is obtained upon registration of a new GitHub OAuth
# application here: https://github.com/settings/applications/new
client_id = "<your client key>"
client_secret = "<your client secret>"
authorization_base_url = 'https://github.com/login/oauth/authorize'
token_url = 'https://github.com/login/oauth/access_token'


@app.route("/")
def demo():
    """Step 1: User Authorization.

    Redirect the user/resource owner to the OAuth provider (i.e. Github)
    using an URL with a few key OAuth parameters.
    """
    github = OAuth2Session(client_id)
    authorization_url, state = github.authorization_url(authorization_base_url)

    # State is used to prevent CSRF, keep this for later.
    session['oauth_state'] = state
    return redirect(authorization_url)


# Step 2: User authorization, this happens on the provider.

@app.route("/callback", methods=["GET"])
def callback():
    """ Step 3: Retrieving an access token.

    The user has been redirected back from the provider to your registered
    callback URL. With this redirection comes an authorization code included
    in the redirect URL. We will use that to obtain an access token.
    """

    github = OAuth2Session(client_id, state=session['oauth_state'])
    token = github.fetch_token(token_url, client_secret=client_secret,
                               authorization_response=request.url)

    # At this point you can fetch protected resources but lets save
```

(continues on next page)

(continued from previous page)

```
    # the token and show how this is done from a persisted token
    # in /profile.
    session['oauth_token'] = token

    return redirect(url_for('.profile'))


@app.route("/profile", methods=["GET"])
def profile():
    """Fetching a protected resource using an OAuth 2 token.
    """
    github = OAuth2Session(client_id, token=session['oauth_token'])
    return jsonify(github.get('https://api.github.com/user').json())


if __name__ == "__main__":
    # This allows us to use a plain HTTP callback
    os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = "1"

    app.secret_key = os.urandom(24)
    app.run(debug=True)
```

This example is lovingly borrowed from this gist.

N.B: You should note that Oauth2 works through SSL layer. If your server is not parametrized to allow HTTPS, the *fetch_token* method will raise an **oauthlib.oauth2.rfc6749.errors.InsecureTransportError**. Most people don't set SSL on their server while testing and that is fine. You can disable this check in two ways:

1. By setting an environment variable.

```
export OAUTHLIB_INSECURE_TRANSPORT=1
```

2. Equivalent to above you can set this in Python (if you have problems setting environment variables)

```
# Somewhere in webapp_example.py, before the app.run for example
import os
os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = '1'
```

3.3.10 Refreshing tokens in OAuth 2

OAuth 2 providers may allow you to refresh access tokens using refresh tokens. Commonly, only clients that authenticate may refresh tokens, e.g. web applications but not javascript clients. The provider will mention whether they allow token refresh in their API documentation and if you see a "refresh_token" in your token response you are good to go.

This example shows how a simple web application (using the Flask web framework) can refresh Google OAuth 2 tokens. It should be trivial to transfer to any other web framework and provider.

```
from pprint import pformat
from time import time

from flask import Flask, request, redirect, session, url_for
from flask.json import jsonify
import requests
from requests_oauthlib import OAuth2Session
```

(continues on next page)

(continued from previous page)

```
app = Flask(__name__)

# This information is obtained upon registration of a new Google OAuth
# application at https://code.google.com/apis/console
client_id = "<your client key>"
client_secret = "<your client secret>"
redirect_uri = 'https://your.registered/callback'

# Uncomment for detailed oauthlib logs
#import logging
#import sys
#log = logging.getLogger('oauthlib')
#log.addHandler(logging.StreamHandler(sys.stdout))
#log.setLevel(logging.DEBUG)

# OAuth endpoints given in the Google API documentation
authorization_base_url = "https://accounts.google.com/o/oauth2/auth"
token_url = "https://accounts.google.com/o/oauth2/token"
refresh_url = token_url # True for Google but not all providers.
scope = [
    "https://www.googleapis.com/auth/userinfo.email",
    "https://www.googleapis.com/auth/userinfo.profile",
]

@app.route("/")
def demo():
    """Step 1: User Authorization.

    Redirect the user/resource owner to the OAuth provider (i.e. Google)
    using an URL with a few key OAuth parameters.
    """
    google = OAuth2Session(client_id, scope=scope, redirect_uri=redirect_uri)
    authorization_url, state = google.authorization_url(authorization_base_url,
        # offline for refresh token
        # force to always make user click authorize
        access_type="offline", prompt="select_account")

    # State is used to prevent CSRF, keep this for later.
    session['oauth_state'] = state
    return redirect(authorization_url)


# Step 2: User authorization, this happens on the provider.
@app.route("/callback", methods=["GET"])
def callback():
    """ Step 3: Retrieving an access token.

    The user has been redirected back from the provider to your registered
    callback URL. With this redirection comes an authorization code included
    in the redirect URL. We will use that to obtain an access token.
    """

    google = OAuth2Session(client_id, redirect_uri=redirect_uri,
                           state=session['oauth_state'])
    token = google.fetch_token(token_url, client_secret=client_secret,
                               authorization_response=request.url)
```

(continues on next page)

(continued from previous page)

```
    # We use the session as a simple DB for this example.
    session['oauth_token'] = token

    return redirect(url_for('.menu'))


@app.route("/menu", methods=["GET"])
def menu():
    """"""
    return """
    <h1>Congratulations, you have obtained an OAuth 2 token!</h1>
    <h2>What would you like to do next?</h2>
    <ul>
        <li><a href="/profile"> Get account profile</a></li>
        <li><a href="/automatic_refresh"> Implicitly refresh the token</a></li>
        <li><a href="/manual_refresh"> Explicitly refresh the token</a></li>
        <li><a href="/validate"> Validate the token</a></li>
    </ul>

    <pre>
    %s
    </pre>
    """ % pformat(session['oauth_token'], indent=4)


@app.route("/profile", methods=["GET"])
def profile():
    """Fetching a protected resource using an OAuth 2 token.
    """
    google = OAuth2Session(client_id, token=session['oauth_token'])
    return jsonify(google.get('https://www.googleapis.com/oauth2/v1/userinfo').json())


@app.route("/automatic_refresh", methods=["GET"])
def automatic_refresh():
    """Refreshing an OAuth 2 token using a refresh token.
    """
    token = session['oauth_token']

    # We force an expiration by setting expired at in the past.
    # This will trigger an automatic refresh next time we interact with
    # Googles API.
    token['expires_at'] = time() - 10

    extra = {
        'client_id': client_id,
        'client_secret': client_secret,
    }

    def token_updater(token):
        session['oauth_token'] = token

    google = OAuth2Session(client_id,
                           token=token,
                           auto_refresh_kwargs=extra,
                           auto_refresh_url=refresh_url,
                           token_updater=token_updater)
```

(continues on next page)

(continued from previous page)

```
    # Trigger the automatic refresh
    jsonify(google.get('https://www.googleapis.com/oauth2/v1/userinfo').json())
    return jsonify(session['oauth_token'])


@app.route("/manual_refresh", methods=["GET"])
def manual_refresh():
    """Refreshing an OAuth 2 token using a refresh token.
    """
    token = session['oauth_token']

    extra = {
        'client_id': client_id,
        'client_secret': client_secret,
    }

    google = OAuth2Session(client_id, token=token)
    session['oauth_token'] = google.refresh_token(refresh_url, **extra)
    return jsonify(session['oauth_token'])

@app.route("/validate", methods=["GET"])
def validate():
    """Validate a token with the OAuth provider Google.
    """
    token = session['oauth_token']

    # Defined at https://developers.google.com/accounts/docs/OAuth2LoginV1
↪#validatingtoken
    validate_url = ('https://www.googleapis.com/oauth2/v1/tokeninfo?'
                    'access_token=%s' % token['access_token'])

    # No OAuth2Session is needed, just a plain GET request
    return jsonify(requests.get(validate_url).json())


if __name__ == "__main__":
    # This allows us to use a plain HTTP callback
    import os
    os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = "1"

    app.secret_key = os.urandom(24)
    app.run(debug=True)
```

3.4 Developer Interface

3.4.1 OAuth 1.0

class async_oauthlib.**OAuth1**(*client_key*, *client_secret=None*, *resource_owner_key=None*, *resource_owner_secret=None*, *callback_uri=None*, *signature_method='HMAC-SHA1'*, *signature_type='AUTH_HEADER'*, *rsa_key=None*, *verifier=None*, *decoding='utf-8'*, *client_class=None*, *force_include_body=False*, ***kwargs*)

Signs the request using OAuth 1 (RFC5849)

client_class

alias of `oauthlib.oauth1.rfc5849.Client`

3.4.2 OAuth 1.0 Session

class async_oauthlib.**OAuth1Session**(*client_key*, *client_secret=None*, *resource_owner_key=None*, *resource_owner_secret=None*, *callback_uri=None*, *signature_method='HMAC-SHA1'*, *signature_type='AUTH_HEADER'*, *rsa_key=None*, *verifier=None*, *client_class=None*, *force_include_body=False*, ***kwargs*)

Request signing and convenience methods for the oauth dance.

What is the difference between OAuth1Session and OAuth1?

OAuth1Session actually uses OAuth1 internally and its purpose is to assist in the OAuth workflow through convenience methods to prepare authorization URLs and parse the various token and redirection responses. It also provide rudimentary validation of responses.

An example of the OAuth workflow using a basic CLI app and Twitter.

```
>>> # Credentials obtained during the registration.
>>> client_key = 'client key'
>>> client_secret = 'secret'
>>> callback_uri = 'https://127.0.0.1/callback'
>>>
>>> # Endpoints found in the OAuth provider API documentation
>>> request_token_url = 'https://api.twitter.com/oauth/request_token'
>>> authorization_url = 'https://api.twitter.com/oauth/authorize'
>>> access_token_url = 'https://api.twitter.com/oauth/access_token'
>>>
>>> oauth_session = OAuth1Session(client_key,client_secret=client_secret,␣
↪callback_uri=callback_uri)
>>>
>>> # First step, fetch the request token.
>>> await oauth_session.fetch_request_token(request_token_url)
{
    'oauth_token': 'kjerht2309u',
    'oauth_token_secret': 'lsdajfh923874',
}
>>>
>>> # Second step. Follow this link and authorize
>>> oauth_session.authorization_url(authorization_url)
'https://api.twitter.com/oauth/authorize?oauth_token=sdf0o9823sjdfsdf&oauth_
↪callback=https%3A%2F%2F127.0.0.1%2Fcallback'
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Third step. Fetch the access token
>>> redirect_response = input('Paste the full redirect URL here.')
>>> oauth_session.parse_authorization_response(redirect_response)
{
    'oauth_token: 'kjerht2309u',
    'oauth_token_secret: 'lsdajfh923874',
    'oauth_verifier: 'w34o8967345',
}
>>> await oauth_session.fetch_access_token(access_token_url)
{
    'oauth_token': 'sdf0o9823sjdfsdf',
    'oauth_token_secret': '2kjshdfp92i34asdasd',
}
>>> # Done. You can now make OAuth requests.
>>> status_url = 'http://api.twitter.com/1/statuses/update.json'
>>> new_status = {'status':  'hello world!'}
>>> await oauth_session.post(status_url, data=new_status)
<Response [200]>
```

authorization_url (*url*, *request_token=None*, ***kwargs*)

Create an authorization URL by appending request_token and optional kwargs to url.

This is the second step in the OAuth 1 workflow. The user should be redirected to this authorization URL, grant access to you, and then be redirected back to you. The redirection back can either be specified during client registration or by supplying a callback URI per request.

Parameters

- **url** – The authorization endpoint URL.
- **request_token** – The previously obtained request token.
- **kwargs** – Optional parameters to append to the URL.

Returns The authorization URL with new parameters embedded.

An example using a registered default callback URI.

```
>>> request_token_url = 'https://api.twitter.com/oauth/request_token'
>>> authorization_url = 'https://api.twitter.com/oauth/authorize'
>>> oauth_session = OAuth1Session('client-key', client_secret='secret')
>>> await oauth_session.fetch_request_token(request_token_url)
{
    'oauth_token': 'sdf0o9823sjdfsdf',
    'oauth_token_secret': '2kjshdfp92i34asdasd',
}
>>> oauth_session.authorization_url(authorization_url)
'https://api.twitter.com/oauth/authorize?oauth_token=sdf0o9823sjdfsdf'
>>> oauth_session.authorization_url(authorization_url, foo='bar')
'https://api.twitter.com/oauth/authorize?oauth_token=sdf0o9823sjdfsdf&foo=bar'
```

An example using an explicit callback URI.

```
>>> request_token_url = 'https://api.twitter.com/oauth/request_token'
>>> authorization_url = 'https://api.twitter.com/oauth/authorize'
>>> oauth_session = OAuth1Session('client-key', client_secret='secret',␣
↪callback_uri='https://127.0.0.1/callback')
>>> await oauth_session.fetch_request_token(request_token_url)
```

(continues on next page)

(continued from previous page)

```
{
    'oauth_token': 'sdf0o9823sjdfsdf',
    'oauth_token_secret': '2kjshdfp92i34asdasd',
}
>>> oauth_session.authorization_url(authorization_url)
'https://api.twitter.com/oauth/authorize?oauth_token=sdf0o9823sjdfsdf&oauth_
↪callback=https%3A%2F%2F127.0.0.1%2Fcallback'
```

authorized

Boolean that indicates whether this session has an OAuth token or not. If *self.authorized* is True, you can reasonably expect OAuth-protected requests to the resource to succeed. If *self.authorized* is False, you need the user to go through the OAuth authentication dance before OAuth-protected requests to the resource will succeed.

fetch_access_token(*url*, *verifier=None*, ***kwargs*)

Fetch an access token.

This is the final step in the OAuth 1 workflow. An access token is obtained using all previously obtained credentials, including the verifier from the authorization step.

Note that a previously set verifier will be reset for your convenience, or else signature creation will be incorrect on consecutive requests.

```
>>> access_token_url = 'https://api.twitter.com/oauth/access_token'
>>> redirect_response = 'https://127.0.0.1/callback?oauth_token=kjerht2309uf&
↪oauth_token_secret=lsdajfh923874&oauth_verifier=w34o8967345'
>>> oauth_session = OAuth1Session('client-key', client_secret='secret')
>>> oauth_session.parse_authorization_response(redirect_response)
{
    'oauth_token: 'kjerht2309u',
    'oauth_token_secret: 'lsdajfh923874',
    'oauth_verifier: 'w34o8967345',
}
>>> await oauth_session.fetch_access_token(access_token_url)
{
    'oauth_token': 'sdf0o9823sjdfsdf',
    'oauth_token_secret': '2kjshdfp92i34asdasd',
}
```

fetch_request_token(*url*, *realm=None*, ***kwargs*)

Fetch a request token.

This is the first step in the OAuth 1 workflow. A request token is obtained by making a signed post request to url. The token is then parsed from the application/x-www-form-urlencoded response and ready to be used to construct an authorization url.

Parameters

- **url** – The request token endpoint URL.
- **realm** – A list of realms to request access to.
- **kwargs** – Optional arguments passed to ''post'' function in ''aiohttp.ClientSession''

Returns The response in dict format.

Note that a previously set callback_uri will be reset for your convenience, or else signature creation will be incorrect on consecutive requests.

```
>>> request_token_url = 'https://api.twitter.com/oauth/request_token'
>>> oauth_session = OAuth1Session('client-key', client_secret='secret')
>>> await oauth_session.fetch_request_token(request_token_url)
{
    'oauth_token': 'sdf0o9823sjdfsdf',
    'oauth_token_secret': '2kjshdfp92i34asdasd',
}
```

parse_authorization_response(*url*)

Extract parameters from the post authorization redirect response URL.

Parameters **url** – The full URL that resulted from the user being redirected back from the OAuth provider to you, the client.

Returns A dict of parameters extracted from the URL.

```
>>> redirect_response = 'https://127.0.0.1/callback?oauth_token=kjerht2309uf&
↪oauth_token_secret=lsdajfh923874&oauth_verifier=w34o8967345'
>>> oauth_session = OAuth1Session('client-key', client_secret='secret')
>>> oauth_session.parse_authorization_response(redirect_response)
{
    'oauth_token: 'kjerht2309u',
    'oauth_token_secret: 'lsdajfh923874',
    'oauth_verifier: 'w34o8967345',
}
```

rebuild_auth(*prepared_request*, *response*)

When being redirected we should always strip Authorization header, since nonce may not be reused as per OAuth spec.

3.4.3 OAuth 2.0

class async_oauthlib.**OAuth2**(*client_id=None*, *client=None*, *token=None*)

Adds proof of authorization (OAuth2 token) to the request.

class async_oauthlib.**TokenUpdated**(*token*)

3.4.4 OAuth 2.0 Session

class async_oauthlib.**OAuth2Session**(*client_id=None*, *client=None*, *auto_refresh_url=None*, *auto_refresh_kwargs=None*, *scope=None*, *redirect_uri=None*, *token=None*, *state=None*, *token_updater=None*, ***kwargs*)

Versatile OAuth 2 extension to `requests.Session`.

Supports any grant type adhering to `oauthlib.oauth2.Client` spec including the four core OAuth 2 grants.

Can be used to create authorization urls, fetch tokens and access protected resources using the `requests.Session` interface you are used to.

- `oauthlib.oauth2.WebApplicationClient` (default): Authorization Code Grant
- `oauthlib.oauth2.MobileApplicationClient`: Implicit Grant
- `oauthlib.oauth2.LegacyApplicationClient`: Password Credentials Grant
- `oauthlib.oauth2.BackendApplicationClient`: Client Credentials Grant

Note that the only time you will be using Implicit Grant from python is if you are driving a user agent able to obtain URL fragments.

authorization_url(*url*, *state=None*, ***kwargs*)
Form an authorization URL.

Parameters

- **url** – Authorization endpoint url, must be HTTPS.
- **state** – An optional state string for CSRF protection. If not given it will be generated for you.
- **kwargs** – Extra parameters to include.

Returns authorization_url, state

authorized
Boolean that indicates whether this session has an OAuth token or not. If *self.authorized* is True, you can reasonably expect OAuth-protected requests to the resource to succeed. If *self.authorized* is False, you need the user to go through the OAuth authentication dance before OAuth-protected requests to the resource will succeed.

fetch_token(*token_url*, *code=None*, *authorization_response=None*, *body=''*, *auth=None*, *username=None*, *password=None*, *method='POST'*, *force_querystring=False*, *timeout=None*, *headers=None*, *verify_ssl=True*, *proxy=None*, *include_client_id=None*, *client_secret=None*, *ssl_context=None*, ***kwargs*)
Generic method for fetching an access token from the token endpoint.

If you are using the MobileApplicationClient you will want to use *token_from_fragment* instead of *fetch_token*.

The current implementation enforces the RFC guidelines.

Parameters

- **token_url** – Token endpoint URL, must use HTTPS.
- **code** – Authorization code (used by WebApplicationClients).
- **authorization_response** – Authorization response URL, the callback URL of the request back to you. Used by WebApplicationClients instead of code.
- **body** – Optional application/x-www-form-urlencoded body to add the include in the token request. Prefer kwargs over body.
- **auth** – An auth tuple or method as accepted by *requests*.
- **username** – Username required by LegacyApplicationClients to appear in the request body.
- **password** – Password required by LegacyApplicationClients to appear in the request body.
- **method** – The HTTP method used to make the request. Defaults to POST, but may also be GET. Other methods should be added as needed.
- **force_querystring** – If True, force the request body to be sent in the querystring instead.
- **timeout** – Timeout of the request in seconds.
- **headers** – Dict to default request headers with.
- **verify_ssl** – Verify SSL certificate.

- **proxy** – The *proxy* argument is passed onto *aiohttp*.
- **include_client_id** – Should the request body include the *client_id* parameter. Default is *None*, which will attempt to autodetect. This can be forced to always include (True) or never include (False).
- **client_secret** – The *client_secret* paired to the *client_id*. This is generally required unless provided in the *auth* tuple. If the value is *None*, it will be omitted from the request, however if the value is an empty string, an empty string will be sent.
- **ssl_context** – The ssl context being passed to *aiohttp* ssl_context.
- **kwargs** – Extra parameters to include in the token request.

Returns A token dict

new_state()
Generates a state string to be used in authorizations.

refresh_token(*token_url*, *refresh_token=None*, *body=''*, *auth=None*, *timeout=None*, *headers=None*, *verify_ssl=True*, *proxy=None*, ***kwargs*)
Fetch a new access token using a refresh token.

Parameters

- **token_url** – The token endpoint, must be HTTPS.
- **refresh_token** – The refresh_token to use.
- **body** – Optional application/x-www-form-urlencoded body to add the include in the token request. Prefer kwargs over body.
- **auth** – An auth tuple or method as accepted by *requests*.
- **timeout** – Timeout of the request in seconds.
- **headers** – A dict of headers to be used by *requests*.
- **verify_ssl** – Verify SSL certificate.
- **proxy** – The *proxy* argument will be passed to *requests*.
- **kwargs** – Extra parameters to include in the token request.

Returns A token dict

register_compliance_hook(*hook_type*, *hook*)
Register a hook for request/response tweaking.

Available hooks are: access_token_response invoked before token parsing. refresh_token_response invoked before refresh token parsing. protected_request invoked before making a request.

If you find a new hook is needed please send a GitHub PR request or open an issue.

request(*method*, *url*, *data=None*, *headers=None*, *withhold_token=False*, *client_id=None*, *client_secret=None*, ***kwargs*)
Intercept all requests and add the OAuth 2 token if present.

token_from_fragment(*authorization_response*)
Parse token from the URI fragment, used by MobileApplicationClients.

Parameters **authorization_response** – The full URL of the redirect back to you

Returns A token dict

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index