OAuthLib Documentation

Release 2.0.2

Idan Gazit and the Python Community

Aug 02, 2017

Contents

If you can't find what you need or have suggestions for improvement, don't hesitate to open a new issue on GitHub!

Check out *Reporting bugs how-to* for details on how to be an awesome bug reporter.

For news and discussions please head over to our G+ OAuthLib community.

CHAPTER 1

Installing OAuthLib

The recommended way to install OAuthLib is from PyPI but if you are running into a bug or want to try out recently implemented features you will want to try installing directly from the GitHub master branch.

For various reasons you may wish to install using your OS packaging system and install instructions for a few are shown below. Please send a PR to add a missing one.

Latest release on PYPI

```
pip install oauthlib
```

Bleeding edge from GitHub master

```
pip install -e git+https://github.com/idan/oauthlib.git#egg=oauthlib
```

Debian and derivatives like Ubuntu, Mint, etc.

```
apt-get install python-oauthlib
apt-get install python3-oauthlib
```

Redhat and Fedora

```
yum install python-oauthlib
yum install python3-oauthlib
```

openSUSE

```
zypper in python-oauthlib
zypper in python3-oauthlib
```

Gentoo

```
emerge oauthlib
```

Arch

```
pacman -S python-oauthlib
pacman -S python2-oauthlib
```

FreeBSD

```
pkg_add -r security/py-oauthlib/
```

CHAPTER 2

Frequently asked questions

How do I enable logging for OAuthLib?

See *Reporting bugs how-to*.

What parts of OAuth 1 & 2 are supported?

See *Supported features and platforms*.

OAuth 1 with RSA-SHA1 signatures says "could not import cryptography". What should I do?

Install cryptography via pip.

```
$ pip install cryptography
```

OAuth 2 ServiceApplicationClient and OAuth 1 with RSA-SHA1 signatures say "could not import jwt". What should I do?

Install pyjwt and cryptography with pip.

```
$ pip install pyjwt cryptography
```

What does ValueError *Only unicode objects are escapable. Got one of type X.* mean?

OAuthLib uses unicode everywhere and when creating a OAuth 1 signature a number of parameters need to be percent encoded (aka escaped). At least one parameter could not be encoded. Usually because *None* or a non UTF-8 encoded string was supplied.

What does ValueError *Error trying to decode a non urlencoded string* mean?

You are trying to decode a response which is not properly encoded, e.g. include non percent encoded characters such as £. Which could be because it has already been decoded by your web framework.

If you believe it contains characters that should be excempt from this check please open an issue and state why.

What is the difference between a client and a consumer?

None, they both refer to the third-party accessing protected resources from an OAuth provider on behalf of a user. In order to do so they have to obtain authorization from said user, which is what the *Auth* in *OAuth* stands for.

How do I use OAuthLib with Google, Twitter and other providers?

Most people will be using OAuthLib indirectly. Clients will want to look at requests-oauthlib.

How do I use OAuthlib as a provider with Django, Flask and other web frameworks?

Providers using Django should seek out django-oauth-toolkit and those using Flask flask-oauthlib. For other frameworks, please get in touch by opening a GitHub issue, on G+ or on IRC #oauthlib irc.freenode.net.

What is the difference between authentication and authorization?

See difference.

Very briefly, what is the biggest difference between OAuth 1 and 2?

OAuth 2 is much simpler since it requires the use of TLS whereas OAuth 1 had the requirement to work securely without TLS. To be secure without TLS OAuth 1 required each request to be signed which can be cumbersome.

Some argue OAuth 2 is worse than 1, is that true?

Correctly implemented, OAuth 2 is better in many ways than OAuth 1. Getting it right is not trivial and a task OAuthLib aims to help make simple.

CHAPTER 3

Supported features and platforms

OAuth 1 is fully supported per the RFC for both clients and providers. Extensions and variations that are outside the spec are not supported.

- HMAC-SHA1, RSA-SHA1 and plaintext signatures.
- Signature placement in header, url or body.

OAuth 2 client and provider support for

- Authorization Code Grant
- Implicit Grant
- Client Credentials Grant
- Resource Owner Password Credentials Grant
- Refresh Tokens
- Bearer Tokens
- Draft MAC tokens
- Token Revocation
- OpenID Connect Authentication

with support for SAML2 and JWT tokens, dynamic client registration and more to come.

Supported platforms

OAuthLib is mainly developed/tested on 64 bit Linux but works on Unix (incl. OS X) and Windows as well. Unless you are using the RSA features of OAuth 1 you should be able to use OAuthLib on any platform that supports Python. If you use RSA you are limited to the platforms supported by cryptography.

CHAPTER 4

Reporting bugs how-to

Bugs are reported by opening a new Github issue and you should never hesitate to do so. Indeed, please open an issue if the documentation is unclear, you think the API is unintuitive or if you just want some help using the library.

OAuthLib strive to have helpful exception messages and if you run into a case where that is not true please let us know!

When reporting bugs, especially when they are hard or impossible to reproduce, it is useful to include logging output. You can enable logging for all oauthlib modules by adding a logger to the *oauthlib* namespace.

```
import logging
import sys
log = logging.getLogger('oauthlib')
log.addHandler(logging.StreamHandler(sys.stdout))
log.setLevel(logging.DEBUG)
```

If you are using a library that builds upon OAuthLib please also enable the logging for their modules, e.g. for *requests-oauthlib*

```
log = logging.getLogger('requests-oauthlib')
log.addHandler(logging.StreamHandler(sys.stdout))
log.setLevel(logging.DEBUG)
```

Unfortunately we can't always respond quickly to issues and to help us help you please try and include steps to reproduce the issue. A short example can go far, e.g. instead of

```
# oauthlib crashes when trying to sign foobar urls.
```

aim for

```
# OAuth 1 Clients raise a value error for the example below
from oauthlib.oauth1 import Client
client = Client('client-id')
headers = {'Content-Type': 'application/x-www-form-urlencoded'}
body = 'hello world'
client.sign('https://foo.bar', headers=headers, body=body)
```

An example like this immediately tells us two things

1. You might want to have the body sign but it was unclear that it needs to be properly encoded first.
2. You might not want the body signed but follow an example where the header was provided and you were not sure if you could simply skip supplying the header.

The root cause could certainly be much more complicated but in either case steps to reproduce allow us to speculate as to what might cause the problem and lower the number of round trips needed to find a solution.

CHAPTER 5

Contributing

Setup

Fork on GitHub

Before you do anything else, login/signup on GitHub and fork OAuthLib from the GitHub project.

Clone your fork locally

If you have git-scm installed, you now clone your git repo using the following command-line argument where <my-github-name> is your account name on GitHub:

```
git clone git@github.com/<my-github-name>/oauthlib.git
```

Issues!

The list of outstanding OAuthLib feature requests and bugs can be found on our on our GitHub issue tracker. Pick an unassigned issue that you think you can accomplish, add a comment that you are attempting to do it, and shortly your own personal label matching your GitHub ID will be assigned to that issue.

Feel free to propose issues that aren't described!

Setting up topic branches and generating pull requests

While it's handy to provide useful code snippets in an issue, it is better for you as a developer to submit pull requests. By submitting pull request your contribution to OpenComparison will be recorded by Github.

In git it is best to isolate each topic or feature into a "topic branch". While individual commits allow you control over how small individual changes are made to the code, branches are a great way to group a set of commits all related to one feature together, or to isolate different efforts when you might be working on multiple topics at the same time.

While it takes some experience to get the right feel about how to break up commits, a topic branch should be limited in scope to a single `issue` as submitted to an issue tracker.

Also since GitHub pegs and syncs a pull request to a specific branch, it is the **ONLY** way that you can submit more than one fix at a time. If you submit a pull from your master branch, you can't make any more commits to your master without those getting added to the pull.

To create a topic branch, its easiest to use the convenient `-b` argument to `git checkout`:

```
git checkout -b fix-broken-thing
Switched to a new branch 'fix-broken-thing'
```

You should use a verbose enough name for your branch so it is clear what it is about. Now you can commit your changes and regularly merge in the upstream master as described below.

When you are ready to generate a pull request, either for preliminary review, or for consideration of merging into the project you must first push your local topic branch back up to GitHub:

```
git push origin fix-broken-thing
```

Now when you go to your fork on GitHub, you will see this branch listed under the "Source" tab where it says "Switch Branches". Go ahead and select your topic branch from this list, and then click the "Pull request" button.

Here you can add a comment about your branch. If this in response to a submitted issue, it is good to put a link to that issue in this initial comment. The repo managers will be notified of your pull request and it will be reviewed (see below for best practices). Note that you can continue to add commits to your topic branch (and push them up to GitHub) either if you see something that needs changing, or in response to a reviewer's comments. If a reviewer asks for changes, you do not need to close the pull and reissue it after making changes. Just make the changes locally, push them to GitHub, then add a comment to the discussion section of the pull request.

Pull upstream changes into your fork

It is critical that you pull upstream changes from master into your fork on a regular basis. Nothing is worse than putting in a days of hard work into a pull request only to have it rejected because it has diverged too far from master.

To pull in upstream changes:

```
git remote add upstream https://github.com/idan/oauthlib.git
git fetch upstream
```

Check the log to be sure that you actually want the changes, before merging:

```
git log upstream/master
```

Then merge the changes that you fetched:

```
git merge upstream/master
```

For more info, see http://help.github.com/fork-a-repo/

How to get your pull request accepted

We want your submission. But we also want to provide a stable experience for our users and the community. Follow these rules and you should succeed without a problem!

Run the tests!

Before you submit a pull request, please run the entire OAuthLib test suite from the project root via:

```
$ python -m unittest discover
```

The first thing the core committers will do is run this command. Any pull request that fails this test suite will be **rejected**.

Testing multiple versions of Python

OAuthLib supports Python 2.6, 2.7, 3.2, 3.3 and experimentally PyPy. Testing all versions conveniently can be done using Tox.

```
$ tox
```

Tox requires you to have virtualenv installed as well as respective python version. For Ubuntu you can easily install all after adding one ppa.

```
$ sudo add-apt-repository ppa:fkrull/deadsnakes
$ sudo apt-get update
$ sudo apt-get install python2.6 python2.6-dev
$ sudo apt-get install python2.7 python2.7-dev
$ sudo apt-get install python3.2 python3.2-dev
$ sudo apt-get install python3.3 python3.3-dev
$ sudo apt-get install pypy pypy-dev
```

If you add code you need to add tests!

We've learned the hard way that code without tests is undependable. If your pull request reduces our test coverage because it lacks tests then it will be **rejected**.

Also, keep your tests as simple as possible. Complex tests end up requiring their own tests. We would rather see duplicated assertions across test methods than cunning utility methods that magically determine which assertions are needed at a particular stage. Remember: *Explicit is better than implicit*.

Don't mix code changes with whitespace cleanup

If you change two lines of code and correct 200 lines of whitespace issues in a file the diff on that pull request is functionally unreadable and will be **rejected**. Whitespace cleanups need to be in their own pull request.

Keep your pull requests limited to a single issue

OauthLib pull requests should be as small/atomic as possible. Large, wide-sweeping changes in a pull request will be **rejected**, with comments to isolate the specific code in your pull request. Some examples:

1. If you are making spelling corrections in the docs, don't modify any Python code.
2. If you are adding a new module don't '*cleanup*' other modules. That cleanup in another pull request.
3. Changing any attributes of a module, such as permissions on a file should be in its own pull request with explicit reasons why.

Follow PEP-8 and keep your code simple!

Memorize the Zen of Python:

```
>>> python -c 'import this'
```

Please keep your code as clean and straightforward as possible. When we see more than one or two functions/methods starting with *_my_special_function* or things like *__builtins__.object = str* we start to get worried. Rather than try and figure out your brilliant work we'll just **reject** it and send along a request for simplification.

Furthermore, the pixel shortage is over. We want to see:

- *package* instead of *pkg*
- *grid* instead of *g*
- *my_function_that_does_things* instead of *mftdt*

How pull requests are checked, tested, and done

First we pull the code into a local branch:

```
git remote add <submitter-github-name> git@github.com:<submitter-github-name>/
↪opencomparison.git
git fetch <submitter-github-name>
git checkout -b <branch-name> <submitter-github-name>/<branch-name>
```

Then we run the tests:

```
python -m unittest discover
```

We finish with a non-fastforward merge (to preserve the branch history) and push to GitHub:

```
git checkout master
git merge --no-ff <branch-name>
git push upstream master
```

CHAPTER 6

Release process

OAuthLib has got to a point where quite a few libraries and users depend on it. Because of this a more careful release procedure will be introduced to make sure all these lovely projects don't suddenly break.

When approaching a release we will run the unittests for a set of downstream libraries using the unreleased version of OAuthLib. If OAuthLib is the cause of failing tests we will either

1. Find a way to introduce the change without breaking downstream. However, this is not always the best long term option.
2. Report the breaking change in the affected projects issue tracker or through Github mentions in a "master" issue on OAuthLib if many projects are affected.

Ideally, this process will allow rapid and graceful releases but in the case of downstream projects remaining in a broken stage for long we will simply advice they lock the oauthlib version in `setup.py` and release anyway.

Unittests might not be enough and as an extra measure we will create an OAuthLib release issue on Github at least 2 days prior to release detailing the changes and pings the primary contacts for each downstream project. Please respond within those 2 days if you have major concerns.

How to get on the notifcations list

Which projects and the instructions for testing each will be defined in OAuthLibs `Makefile`. To add your project, simply open a pull request or notify that you would like to be added by opening a github issue. Please also include github users which can be addressed in Github mentions as primary contacts for the project.

When is the next release?

Releases have been sporadic at best and I don't think that will change soon. However, if you think it's time for a new release don't hesitate to open a new issue asking about it.

A note on versioning

Historically OAuthLib has not been very good at semantic versioning but that will change after the 1.0.0 release due late 2014. After that poing any major digit release (e.g. 2.0.0) may introduce non backwards compatible changes. Minor point (1.1.0) releases will introduce non API breaking new features and changes. Bug releases (1.0.1) will include minor fixes that needs to be released quickly (e.g. after a bigger release unintentionally introduced a bug).

CHAPTER 7

OAuth 1 versus OAuth 2

This is intended to serve as a quick guide to which OAuth version might suit your needs best. The target audience are providers contemplating which workflows to offer their clients but clients curious to which workflow to use should be able to get some help too.

Before choosing it is important to understand a fundamental issue with client - server security. **It is technically impossible to store secrets on machines out of your control, such as a users desktop or phone.** Without the ability to secure a secret the ability to authenticate is lost. Because of this the provider has no way of knowing whether a request from such a client is legitimate or from a malicious party. Great care should be taken to restrict non authenticated clients access to resources appropriately.

When to offer which OAuth workflow

- Your clients reside in secure environments (i.e. able to keep secrets), able to use SSL/TLS and you are willing to risk unknowingly granting access to your users resources to a malicious third party which has stolen tokens (but not authentication secrets) from one of your clients.

  (Provider) Offer *Authorization Code Grant*. Impact can be limited by not providing refresh tokens. Default in *WebApplicationServer*.

  (Client) Use *Web Application Client*.

- Similar to above, but you are unwilling to risk malicious access based on stolen tokens alone.

  (Provider) Offer *OAuth 1*.

  (Client) Use *OAuth 1 Client*.

- Your clients reside in user controlled devices with the ability to authorize through a web based workflow. This workflow is inherently insecure, restrict the privileges associated with tokens accordingly.

  (Provider) Offer *Implicit Grant*. Default in *MobileApplicationServer*.

  (Client) Use *Mobile Application Client*.

- Similar to above but without the ability to use web authorization. These clients must have a strong trust relationship with the users although they offer no additional security.

  (Provider) Offer non authenticated *Resource Owner Password Credentials Grant*. Default in *LegacyApplicationServer*.

(Client) Use *Legacy Application Client*.

- Your clients are transitioning from using usernames/passwords to interact with your API to using OAuth tokens but for various reasons don't wish to use the web based authorization workflow. The clients reside in secure environments and have a strong trust relationship with their users.

  (Provider) Offer authenticated *Resource Owner Password Credentials Grant*. Default in *LegacyApplicationServer*.

  (Client) Use *Legacy Application Client*.

- You wish to run an internal, highly trusted, job acting on protected resources but not interacting with users.

  (Provider) Offer *Client Credentials Grant*. Default in *BackendApplicationServer*.

  (Client) Use *Backend Application Client*.

CHAPTER 8

OAuth 1.0

Using the Client

Are you using requests?

If you are, then you should take a look at requests-oauthlib which has several examples of how to use OAuth1 with requests.

Signing a request with an HMAC-SHA1 signature (most common)

See requests-oauthlib for more detailed examples of going through the OAuth workflow. In a nutshell you will be doing three types of requests, to obtain a request token, to obtain an access token and to access a protected resource.

Obtaining a request token will require client key and secret which are provided to you when registering a client with the OAuth provider:

```
client = oauthlib.oauth1.Client('client_key', client_secret='your_secret')
uri, headers, body = client.sign('http://example.com/request_token')
```

You will then need to redirect to the authorization page of the OAuth provider, which will later redirect back with a verifier and a token secret parameter appended to your callback url. These will be used in addition to the credentials from before when obtaining an access token:

```
client = oauthlib.oauth1.Client('client_key', client_secret='your_secret',
    resource_owner_key='the_request_token', resource_owner_secret='the_
↪request_token_secret',
    verifier='the_verifier')
uri, headers, body = client.sign('http://example.com/access_token')
```

The provider will now give you an access token and a new token secret which you will use to access protected resources:

```
client = oauthlib.oauth1.Client('client_key', client_secret='your_secret',
    resource_owner_key='the_access_token', resource_owner_secret='the_access_
↪token_secret')
uri, headers, body = client.sign('http://example.com/protected_resource')
```

Unicode Everywhere

Starting with 0.3.5 OAuthLib supports automatic conversion to unicode if you supply input in utf-8 encoding. If you are using another encoding you will have to make sure to convert all input to unicode before passing it to OAuthLib. Note that the automatic conversion is limited to the use of oauthlib.oauth1.Client.

Request body

The OAuth 1 spec only covers signing of x-www-url-formencoded information. If you are sending some other kind of data in the body (say, multipart file uploads), these don't count as a body for the purposes of signing. Don't provide the body to Client.sign() if it isn't x-www-url-formencoded data.

For convenience, you can pass body data in one of three ways:

- a dictionary
- an iterable of 2-tuples
- a properly-formatted x-www-url-formencoded string

RSA Signatures

OAuthLib supports 'RSA-SHA1' signatures, but does not install the PyJWT or cryptography dependencies by default. OAuthLib uses the PyJWT package to smooth out its internal code. The cryptography package is much better supported on Windows and Mac OS X than PyCrypto, and simpler to install. Users can install PyJWT and cryptography using pip:

```
pip install pyjwt cryptography
```

When you have cryptography and PyJWT installed, using RSA signatures is similar to HMAC but differ in a few aspects. RSA signatures does not make use of client secrets nor resource owner secrets (token secrets) and requires you to specify the signature type when constructing a client:

```
client = oauthlib.oauth1.Client('your client key',
    signature_method=oauthlib.oauth1.SIGNATURE_RSA,
    resource_owner_key='a token you have obtained',
    rsa_key=open('your_private_key.pem').read())
```

Plaintext signatures

OAuthLib supports plaintext signatures and they are identical in use to HMAC-SHA1 signatures except that you will need to set the signature_method when constructing Clients:

```
client = oauthlib.oauth1.Client('your client key',
    client_secret='your secret',
    resource_owner_key='a token you have obtained',
    resource_owner_secret='a token secret',
    signature_method=oauthlib.oauth1.SIGNATURE_PLAINTEXT)
```

Where to put the signature? Signature types

OAuth 1 commonly use the Authorization header to pass the OAuth signature and other OAuth parameters. This is the default setting in Client and need not be specified. However you may also use the request url query or the request body to pass the parameters. You can specify this location using the signature_type constructor parameter, as shown below:

```
>>> # Embed in Authorization header (recommended)
>>> client = oauthlib.oauth1.Client('client_key',
        signature_type=SIGNATURE_TYPE_AUTH_HEADER,
    )

>>> uri, headers, body = client.sign('http://example.com/path?query=hello')
>>> headers
{u'Authorization': u'OAuth oauth_nonce="107143098223781054691360095427",␣
↪oauth_timestamp="1360095427", oauth_version="1.0", oauth_signature_method=
↪"HMAC-SHA1", oauth_consumer_key="client_key", oauth_signature=
↪"86gpxY1DUXSBRRyWnRNJekeWEzw%3D"'}

>>> # Embed in url query
>>> client = oauthlib.oauth1.Client('client_key',
        signature_type=SIGNATURE_TYPE_QUERY,
    )

>>> uri, headers, body = client.sign('http://example.com/path?query=hello')
>>> uri
http://example.com/path?query=hello&oauth_
↪nonce=97599600646423262881360095509&oauth_timestamp=1360095509&oauth_
↪version=1.0&oauth_signature_method=HMAC-SHA1&oauth_consumer_key=client_key&
↪oauth_signature=VQAib%2F4uRPwfVmCZkgSE3q2p7zU%3D

>>> # Embed in body
>>> client = oauthlib.oauth1.Client('client_key',
        signature_type=SIGNATURE_TYPE_BODY,
    )

>>> # Please set content-type to application/x-www-form-urlencoded
>>> headers = {'Content-Type':oauthlib.oauth1.CONTENT_TYPE_FORM_URLENCODED}
>>> uri, headers, body = client.sign('http://example.com/path?query=hello',
                                     headers=headers)
>>> body
u'oauth_nonce=14809240824815328251136009572 2&oauth_timestamp=1360095722&
↪oauth_version=1.0&oauth_signature_method=HMAC-SHA1&oauth_consumer_
↪key=client_key&oauth_signature=5IKjrRKU3%2FIduI9UumVI%2FbQ0Hv0%3D'
```

Creating a Provider

OAuthLib is a framework independent library that may be used with any web framework. That said, there are framework specific helper libraries to make your life easier.

- For Flask there is flask-oauthlib.

If there is no support for your favourite framework and you are interested in providing it then you have come to the right place. OAuthLib can handle the OAuth logic and leave you to support a few framework and setup specific tasks such as marshalling request objects into URI, headers and body arguments as well as provide an interface for a backend to store tokens, clients, etc.

Tutorial Contents

1. Create your datastore models

These models will represent various OAuth specific concepts. There are a few important links between them that the security of OAuth is based on. Below is a suggestion for models and why you need certain properties. There is also example SQLAlchemy model fields which should be straightforward to translate to other ORMs such as Django and the Appengine Datastore.

1.1 User (or Resource Owner)

The user of your site which resources might be access by clients upon authorization from the user. Below is a crude example of a User model, yours is likely to differ and the structure is not important. Neither is how the user authenticates, as long as it does before authorizing:

```
Base = sqlalchemy.ext.declarative.declarative_base()
class ResourceOwner(Base):
    __tablename__ = "users"

    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String)
    email = sqlalchemy.Column(sqlalchemy.String)
    password = sqlalchemy.Column(sqlalchemy.String)
```

1.2 Client (or Consumer)

The client interested in accessing protected resources.

Client Identifier / Consumer key: Required. The identifier the client will use during the OAuth workflow. Structure is up to you and may be a simple UID:

```
client_key = sqlalchemy.Column(sqlalchemy.String)
```

Client secret: Required for HMAC-SHA1 and PLAINTEXT. The secret the client will use when verifying requests during the OAuth workflow. Has to be accesible as plaintext (i.e. not hashed) since it is used to recreate and validate request signatured:

```
client_secret = sqlalchemy.Column(sqlalchemy.String)
```

Client public key: Required for RSA-SHA1. The public key used to verify the signature of requests signed by the clients private key:

```
rsa_key = sqlalchemy.Column(sqlalchemy.String)
```

User: Recommended. It is common practice to link each client with one of your existing users. Whether you do associate clients and users or not, ensure you are able to protect yourself against malicious clients:

```
user = Column(Integer, ForeignKey("users.id"))
```

Realms: Required. The list of realms the client may request access to. While realm use is largely undocumented in the spec you may think of them as very similar to OAuth 2 scopes.:

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
realms = sqlalchemy.Column(sqlalchemy.String)

# You might also want to mark a certain set of scopes as default
# scopes in case the client does not specify any in the authorization
default_realms = sqlalchemy.Column(sqlalchemy.String)
```

Redirect URIs: These are the absolute URIs that a client may use to redirect to after authorization. You should never allow a client to redirect to a URI that has not previously been registered:

```
# You could represent the URIs either as a list of keys or by
# serializing them into a string.
redirect_uris = sqlalchemy.Column(sqlalchemy.String)

# You might also want to mark a certain URI as default in case the
# client does not specify any in the authorization
default_redirect_uri = sqlalchemy.Column(sqlalchemy.String)
```

1.3 Request Token + Verifier

In OAuth 1 workflow the first step is obtaining/providing a request token. This token captures information about the client, its callback uri and realms requested. This step is not present in OAuth2 as these credentials are supplied directly in the authorization step.

When the request token is first created the user is unknown. The user is associated with a request token during the authorization step. After successful authorization the client is presented with a verifier code (should be linked to request token) as a proof of authorization. This verifier code is later used to obtain an access token.

Client: Association with the client to whom the request token was given:

```
client = Column(Integer, ForeignKey("clients.id"))
```

User: Association with the user to which protected resources this token requests access:

```
user = Column(Integer, ForeignKey("users.id"))
```

Realms: Realms to which the token is bound. Attempt to access protected resources outside these realms will be denied:

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
realms = sqlalchemy.Column(sqlalchemy.String)
```

Redirect URI: The callback URI used to redirect back to the client after user authorization is completed:

```
redirect_uri = sqlalchemy.Column(sqlalchemy.String)
```

Request Token: An unguessable unique string of characters:

```
request_token = sqlalchemy.Column(sqlalchemy.String)
```

Request Token Secret: An unguessable unique string of characters. This is a temporary secret used by the HMAC-SHA1 and PLAINTEXT signature methods when obtaining an access token later:

```
request_token_secret = sqlalchemy.Column(sqlalchemy.String)
```

Authorization Verifier: An unguessable unique string of characters. This code asserts that the user has given the client authorization to access the requested realms. It is initially nil when the client obtains the request token in the first step, and set after user authorization is given in the second step:

```
verifier = sqlalchemy.Column(sqlalchemy.String)
```

1.4 Access Token

Access tokens are provided to clients able to present a valid request token together with its associated verifier. It will allow the client to access protected resources and is normally not associated with an expiration. Although you should consider expiring them as it increases security dramatically.

The user and realms will need to be transferred from the request token to the access token. It is possible that the list of authorized realms is smaller than the list of requested realms. Clients can observe whether this is the case by comparing the *oauth_realms* parameter given in the token reponse. This way of indicating change of realms is backported from OAuth2 scope behaviour and is not in the OAuth 1 spec.

Client: Association with the client to whom the access token was given:

```
client = Column(Integer, ForeignKey("clients.id"))
```

User: Association with the user to which protected resources this token grants access:

```
user = Column(Integer, ForeignKey("users.id"))
```

Realms: Realms to which the token is bound. Attempt to access protected resources outside these realms will be denied:

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
realms = sqlalchemy.Column(sqlalchemy.String)
```

Access Token: An unguessable unique string of characters:

```
access_token = sqlalchemy.Column(sqlalchemy.String)
```

Access Token Secret: An unguessable unique string of characters. This secret is used by the HMAC-SHA1 and PLAINTEXT signature methods when accessing protected resources:

```
access_token_secret = sqlalchemy.Column(sqlalchemy.String)
```

2. Implement a validator

The majority of the work involved in implementing an OAuth 1 provider relates to mapping various validation and persistence methods to a storage backend. The not very accurately named interface you will need to implement is called a *RequestValidator* (name suggestions welcome).

An example of a very basic implementation of the `validate_client_key` method can be seen below:

```
from oauthlib.oauth1 import RequestValidator

# From the previous section on models
from my_models import Client

class MyRequestValidator(RequestValidator):

    def validate_client_key(self, client_key, request):
        try:
            Client.query.filter_by(client_key=client_key).one()
            return True
        except NoResultFound:
            return False
```

The full API you will need to implement is available in the *RequestValidator* section. You might not need to implement all methods depending on which signature methods you wish to support.

Relevant sections include:

Request Validator

class `oauthlib.oauth1.`**`RequestValidator`**

A validator/datastore interaction base class for OAuth 1 providers.

OAuth providers should inherit from RequestValidator and implement the methods and properties outlined below. Further details are provided in the documentation for each method and property.

Methods used to check the format of input parameters. Common tests include length, character set, membership, range or pattern. These tests are referred to as whitelisting or blacklisting. Whitelisting is better but blacklisting can be usefull to spot malicious activity. The following have methods a default implementation:

- check_client_key
- check_request_token
- check_access_token
- check_nonce
- check_verifier
- check_realms

The methods above default to whitelist input parameters, checking that they are alphanumerical and between a minimum and maximum length. Rather than overloading the methods a few properties can be used to configure these methods.

- @safe_characters -> (character set)

- @client_key_length -> (min, max)
- @request_token_length -> (min, max)
- @access_token_length -> (min, max)
- @nonce_length -> (min, max)
- @verifier_length -> (min, max)
- @realms -> [list, of, realms]

Methods used to validate/invalidate input parameters. These checks usually hit either persistent or temporary storage such as databases or the filesystem. See each methods documentation for detailed usage. The following methods must be implemented:

- validate_client_key
- validate_request_token
- validate_access_token
- validate_timestamp_and_nonce
- validate_redirect_uri
- validate_requested_realms
- validate_realms
- validate_verifier
- invalidate_request_token

Methods used to retrieve sensitive information from storage. The following methods must be implemented:

- get_client_secret
- get_request_token_secret
- get_access_token_secret
- get_rsa_key
- get_realms
- get_default_realms
- get_redirect_uri

Methods used to save credentials. The following methods must be implemented:

- save_request_token
- save_verifier
- save_access_token

Methods used to verify input parameters. This methods are used during authorizing request token by user (AuthorizationEndpoint), to check if parameters are valid. During token authorization request is not signed, thus 'validation' methods can not be used. The following methods must be implemented:

- verify_realms
- verify_request_token

To prevent timing attacks it is necessary to not exit early even if the client key or resource owner key is invalid. Instead dummy values should be used during the remaining verification process. It is very important that the dummy client and token are valid input parameters to the methods get_client_secret, get_rsa_key and

get_(access/request)_token_secret and that the running time of those methods when given a dummy value remain equivalent to the running time when given a valid client/resource owner. The following properties must be implemented:

- @dummy_client
- @dummy_request_token
- @dummy_access_token

Example implementations have been provided, note that the database used is a simple dictionary and serves only an illustrative purpose. Use whichever database suits your project and how to access it is entirely up to you. The methods are introduced in an order which should make understanding their use more straightforward and as such it could be worth reading what follows in chronological order.

check_access_token(*request_token*)
Checks that the token contains only safe characters and is no shorter than lower and no longer than upper.

check_client_key(*client_key*)
Check that the client key only contains safe characters and is no shorter than lower and no longer than upper.

check_nonce(*nonce*)
Checks that the nonce only contains only safe characters and is no shorter than lower and no longer than upper.

check_realms(*realms*)
Check that the realm is one of a set allowed realms.

check_request_token(*request_token*)
Checks that the request token contains only safe characters and is no shorter than lower and no longer than upper.

check_verifier(*verifier*)
Checks that the verifier contains only safe characters and is no shorter than lower and no longer than upper.

dummy_access_token
Dummy access token used when an invalid token was supplied.

Returns The dummy access token string.

The dummy access token should be associated with an access token secret such that get_access_token_secret(.., dummy_access_token) returns a valid secret.

This method is used by

- ResourceEndpoint

dummy_client
Dummy client used when an invalid client key is supplied.

Returns The dummy client key string.

The dummy client should be associated with either a client secret, a rsa key or both depending on which signature methods are supported. Providers should make sure that

get_client_secret(dummy_client) get_rsa_key(dummy_client)

return a valid secret or key for the dummy client.

This method is used by

- AccessTokenEndpoint
- RequestTokenEndpoint

•ResourceEndpoint

•SignatureOnlyEndpoint

dummy_request_token
Dummy request token used when an invalid token was supplied.

Returns The dummy request token string.

The dummy request token should be associated with a request token secret such that get_request_token_secret(.., dummy_request_token) returns a valid secret.

This method is used by

•AccessTokenEndpoint

get_access_token_secret(*client_key*, *token*, *request*)
Retrieves the shared secret associated with the access token.

Parameters

- **client_key** – The client/consumer key.
- **token** – The access token string.
- **request** – An oauthlib.common.Request object.

Returns The token secret as a string.

This method must allow the use of a dummy values and the running time must be roughly equivalent to that of the running time of valid values:

```
# Unlikely to be near constant time as it uses two database
# lookups for a valid client, and only one for an invalid.
from your_datastore import AccessTokenSecret
if AccessTokenSecret.has(client_key):
    return AccessTokenSecret.get((client_key, request_token))
else:
    return 'dummy'

# Aim to mimic number of latency inducing operations no matter
# whether the client is valid or not.
from your_datastore import AccessTokenSecret
return ClientSecret.get((client_key, request_token), 'dummy')
```

Note that the returned key must be in plaintext.

This method is used by

•ResourceEndpoint

get_client_secret(*client_key*, *request*)
Retrieves the client secret associated with the client key.

Parameters

- **client_key** – The client/consumer key.
- **request** – An oauthlib.common.Request object.

Returns The client secret as a string.

This method must allow the use of a dummy client_key value. Fetching the secret using the dummy key must take the same amount of time as fetching a secret for a valid client:

```
# Unlikely to be near constant time as it uses two database
# lookups for a valid client, and only one for an invalid.
from your_datastore import ClientSecret
if ClientSecret.has(client_key):
    return ClientSecret.get(client_key)
else:
    return 'dummy'

# Aim to mimic number of latency inducing operations no matter
# whether the client is valid or not.
from your_datastore import ClientSecret
return ClientSecret.get(client_key, 'dummy')
```

Note that the returned key must be in plaintext.

This method is used by

- AccessTokenEndpoint
- RequestTokenEndpoint
- ResourceEndpoint
- SignatureOnlyEndpoint

get_default_realms (*client_key*, *request*)
Get the default realms for a client.

Parameters

- **client_key** – The client/consumer key.
- **request** – An oauthlib.common.Request object.

Returns The list of default realms associated with the client.

The list of default realms will be set during client registration and is outside the scope of OAuthLib.

This method is used by

- RequestTokenEndpoint

get_realms (*token*, *request*)
Get realms associated with a request token.

Parameters

- **token** – The request token string.
- **request** – An oauthlib.common.Request object.

Returns The list of realms associated with the request token.

This method is used by

- AuthorizationEndpoint
- AccessTokenEndpoint

get_redirect_uri (*token*, *request*)
Get the redirect URI associated with a request token.

Parameters

- **token** – The request token string.
- **request** – An oauthlib.common.Request object.

Returns The redirect URI associated with the request token.

It may be desirable to return a custom URI if the redirect is set to "oob". In this case, the user will be redirected to the returned URI and at that endpoint the verifier can be displayed.

This method is used by

•AuthorizationEndpoint

get_request_token_secret(*client_key*, *token*, *request*)
Retrieves the shared secret associated with the request token.

Parameters

- **client_key** – The client/consumer key.
- **token** – The request token string.
- **request** – An oauthlib.common.Request object.

Returns The token secret as a string.

This method must allow the use of a dummy values and the running time must be roughly equivalent to that of the running time of valid values:

```
# Unlikely to be near constant time as it uses two database
# lookups for a valid client, and only one for an invalid.
from your_datastore import RequestTokenSecret
if RequestTokenSecret.has(client_key):
    return RequestTokenSecret.get((client_key, request_token))
else:
    return 'dummy'

# Aim to mimic number of latency inducing operations no matter
# whether the client is valid or not.
from your_datastore import RequestTokenSecret
return ClientSecret.get((client_key, request_token), 'dummy')
```

Note that the returned key must be in plaintext.

This method is used by

•AccessTokenEndpoint

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

Parameters

- **client_key** – The client/consumer key.
- **request** – An oauthlib.common.Request object.

Returns The rsa public key as a string.

This method must allow the use of a dummy client_key value. Fetching the rsa key using the dummy key must take the same amount of time as fetching a key for a valid client. The dummy key must also be of the same bit length as client keys.

Note that the key must be returned in plaintext.

This method is used by

•AccessTokenEndpoint

•RequestTokenEndpoint

- ResourceEndpoint
- SignatureOnlyEndpoint

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

Parameters

- **client_key** – The client/consumer key.
- **request_token** – The request token string.
- **request** – An oauthlib.common.Request object.

Returns None

Per **'Section 2.3'__** of the spec:

"The server MUST (...) ensure that the temporary credentials have not expired or been used before."

This method should ensure that provided token won't validate anymore. It can be simply removing RequestToken from storage or setting specific flag that makes it invalid (note that such flag should be also validated during request token validation).

This method is used by

- AccessTokenEndpoint

save_access_token(*token*, *request*)
Save an OAuth1 access token.

Parameters

- **token** – A dict with token credentials.
- **request** – An oauthlib.common.Request object.

The token dictionary will at minimum include

- `oauth_token` the access token string.
- `oauth_token_secret` the token specific secret used in signing.
- `oauth_authorized_realms` a space separated list of realms.

Client key can be obtained from `request.client_key`.

The list of realms (not joined string) can be obtained from `request.realm`.

This method is used by

- AccessTokenEndpoint

save_request_token(*token*, *request*)
Save an OAuth1 request token.

Parameters

- **token** – A dict with token credentials.
- **request** – An oauthlib.common.Request object.

The token dictionary will at minimum include

- `oauth_token` the request token string.
- `oauth_token_secret` the token specific secret used in signing.
- `oauth_callback_confirmed` the string `true`.

Client key can be obtained from `request.client_key`.

This method is used by

•RequestTokenEndpoint

save_verifier(*token*, *verifier*, *request*)
Associate an authorization verifier with a request token.

Parameters **token** – A request token string.

:param verifier A dictionary containing the oauth_verifier and oauth_token

Parameters **request** – An oauthlib.common.Request object.

We need to associate verifiers with tokens for validation during the access token request.

Note that unlike save_x_token token here is the `oauth_token` token string from the request token saved previously.

This method is used by

•AuthorizationEndpoint

validate_access_token(*client_key*, *token*, *request*)
Validates that supplied access token is registered and valid.

Parameters

- **client_key** – The client/consumer key.
- **token** – The access token string.
- **request** – An oauthlib.common.Request object.

Returns True or False

Note that if the dummy access token is supplied it should validate in the same or nearly the same amount of time as a valid one.

Ensure latency inducing tasks are mimiced even for dummy clients. For example, use:

```
from your_datastore import AccessToken
try:
    return AccessToken.exists(client_key, access_token)
except DoesNotExist:
    return False
```

Rather than:

```
from your_datastore import AccessToken
if access_token == self.dummy_access_token:
    return False
else:
    return AccessToken.exists(client_key, access_token)
```

This method is used by

•ResourceEndpoint

validate_client_key(*client_key*, *request*)
Validates that supplied client key is a registered and valid client.

Parameters

- **client_key** – The client/consumer key.
- **request** – An oauthlib.common.Request object.

Returns True or False

Note that if the dummy client is supplied it should validate in same or nearly the same amount of time as a valid one.

Ensure latency inducing tasks are mimiced even for dummy clients. For example, use:

```
from your_datastore import Client
try:
    return Client.exists(client_key, access_token)
except DoesNotExist:
    return False
```

Rather than:

```
from your_datastore import Client
if access_token == self.dummy_access_token:
    return False
else:
    return Client.exists(client_key, access_token)
```

This method is used by

•AccessTokenEndpoint

•RequestTokenEndpoint

•ResourceEndpoint

•SignatureOnlyEndpoint

validate_realms(*client_key*, *token*, *request*, *uri=None*, *realms=None*)
Validates access to the request realm.

Parameters

- **client_key** – The client/consumer key.
- **token** – A request token string.
- **request** – An oauthlib.common.Request object.
- **uri** – The URI the realms is protecting.
- **realms** – A list of realms that must have been granted to the access token.

Returns True or False

How providers choose to use the realm parameter is outside the OAuth specification but it is commonly used to restrict access to a subset of protected resources such as “photos”.

realms is a convenience parameter which can be used to provide a per view method pre-defined list of allowed realms.

Can be as simple as:

```
from your_datastore import RequestToken
request_token = RequestToken.get(token, None)

if not request_token:
```

```
    return False
return set(request_token.realms).issuperset(set(realms))
```

This method is used by

•ResourceEndpoint

validate_redirect_uri (*client_key*, *redirect_uri*, *request*)
Validates the client supplied redirection URI.

Parameters

- **client_key** – The client/consumer key.
- **redirect_uri** – The URI the client which to redirect back to after authorization is successful.
- **request** – An oauthlib.common.Request object.

Returns True or False

It is highly recommended that OAuth providers require their clients to register all redirection URIs prior to using them in requests and register them as absolute URIs. See CWE-601 for more information about open redirection attacks.

By requiring registration of all redirection URIs it should be straightforward for the provider to verify whether the supplied redirect_uri is valid or not.

Alternatively per Section 2.1 of the spec:

"If the client is unable to receive callbacks or a callback URI has been established via other means, the parameter value MUST be set to "oob" (case sensitive), to indicate an out-of-band configuration."

This method is used by

•RequestTokenEndpoint

validate_request_token (*client_key*, *token*, *request*)
Validates that supplied request token is registered and valid.

Parameters

- **client_key** – The client/consumer key.
- **token** – The request token string.
- **request** – An oauthlib.common.Request object.

Returns True or False

Note that if the dummy request_token is supplied it should validate in the same nearly the same amount of time as a valid one.

Ensure latency inducing tasks are mimiced even for dummy clients. For example, use:

```
from your_datastore import RequestToken
try:
    return RequestToken.exists(client_key, access_token)
except DoesNotExist:
    return False
```

Rather than:

```
from your_datastore import RequestToken
if access_token == self.dummy_access_token:
    return False
else:
    return RequestToken.exists(client_key, access_token)
```

This method is used by

•AccessTokenEndpoint

validate_requested_realms(*client_key*, *realms*, *request*)
Validates that the client may request access to the realm.

Parameters

- **client_key** – The client/consumer key.
- **realms** – The list of realms that client is requesting access to.
- **request** – An oauthlib.common.Request object.

Returns True or False

This method is invoked when obtaining a request token and should tie a realm to the request token and after user authorization this realm restriction should transfer to the access token.

This method is used by

•RequestTokenEndpoint

validate_timestamp_and_nonce(*client_key*, *timestamp*, *nonce*, *request*, *request_token=None*, *access_token=None*)
Validates that the nonce has not been used before.

Parameters

- **client_key** – The client/consumer 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.
- **request** – An oauthlib.common.Request object.

Returns True or False

Per Section 3.3 of the spec.

"A nonce is a random string, uniquely generated by the client to allow the server to verify that a request has never been made before and helps prevent replay attacks when requests are made over a non-secure channel. The nonce value MUST be unique across all requests with the same timestamp, client credentials, and token combinations."

One of the first validation checks that will be made is for the validity of the nonce and timestamp, which are associated with a client key and possibly a token. If invalid then immediately fail the request by returning False. If the nonce/timestamp pair has been used before and you may just have detected a replay attack. Therefore it is an essential part of OAuth security that you not allow nonce/timestamp reuse. Note that this validation check is done before checking the validity of the client and token.:

```
nonces_and_timestamps_database = [
   (u'foo', 1234567890, u'rannoMstrInghere', u'bar')
]

def validate_timestamp_and_nonce(self, client_key, timestamp, nonce,
   request_token=None, access_token=None):

   return ((client_key, timestamp, nonce, request_token or access_token)
            not in self.nonces_and_timestamps_database)
```

This method is used by

- AccessTokenEndpoint
- RequestTokenEndpoint
- ResourceEndpoint
- SignatureOnlyEndpoint

validate_verifier(*client_key*, *token*, *verifier*, *request*)

Validates a verification code.

Parameters

- **client_key** – The client/consumer key.
- **token** – A request token string.
- **verifier** – The authorization verifier string.
- **request** – An oauthlib.common.Request object.

Returns True or False

OAuth providers issue a verification code to clients after the resource owner authorizes access. This code is used by the client to obtain token credentials and the provider must verify that the verifier is valid and associated with the client as well as the resource owner.

Verifier validation should be done in near constant time (to avoid verifier enumeration). To achieve this we need a constant time string comparison which is provided by OAuthLib in `oauthlib.common.safe_string_equals`:

```
from your_datastore import Verifier
correct_verifier = Verifier.get(client_key, request_token)
from oauthlib.common import safe_string_equals
return safe_string_equals(verifier, correct_verifier)
```

This method is used by

- AccessTokenEndpoint

verify_realms(*token*, *realms*, *request*)

Verify authorized realms to see if they match those given to token.

Parameters

- **token** – An access token string.
- **realms** – A list of realms the client attempts to access.
- **request** – An oauthlib.common.Request object.

Returns True or False

This prevents the list of authorized realms sent by the client during the authorization step to be altered to include realms outside what was bound with the request token.

Can be as simple as:

```
valid_realms = self.get_realms(token)
return all((r in valid_realms for r in realms))
```

This method is used by

•AuthorizationEndpoint

verify_request_token(*token*, *request*)
Verify that the given OAuth1 request token is valid.

Parameters

- **token** – A request token string.
- **request** – An oauthlib.common.Request object.

Returns True or False

This method is used only in AuthorizationEndpoint to check whether the oauth_token given in the authorization URL is valid or not. This request is not signed and thus similar `validate_request_token` method can not be used.

This method is used by

•AuthorizationEndpoint

A few important facts regarding OAuth security

- **OAuth without SSL is a Bad Idea™ and it's strongly recommended to use** SSL for all interactions both with your API as well as for setting up tokens. An example of when it's especially bad is when sending POST requests with form data, this data is not accounted for in the OAuth signature and a successfull man-in-the-middle attacker could swap your form data (or files) to whatever he pleases without invalidating the signature. This is an even bigger issue if you fail to check nonce/timestamp pairs for each request, allowing an attacker who intercept your request to replay it later, overriding your initial request. **Server defaults to fail all requests which are not made over HTTPS**, you can explicitly disable this using the enforce_ssl property.
- **Tokens must be random, OAuthLib provides a method for generating** secure tokens and it's packed into `oauthlib.common.generate_token`, use it. If you decide to roll your own, use `random.SystemRandom` which is based on `os.urandom` rather than the default `random` based on the effecient but not truly random Mersenne Twister. Predictable tokens allow attackers to bypass virtually all defences OAuth provides.
- **Timing attacks are real and more than possible if you host your** application inside a shared datacenter. Ensure all `validate_` methods execute in near constant time no matter which input is given. This will be covered in more detail later. Failing to account for timing attacks could **enable attackers to enumerate tokens and successfully guess HMAC secrets**. Note that RSA keys are protected through RSA blinding and are not at risk.
- **Nonce and timestamps must be checked, do not ignore this as it's a** simple and effective way to prevent replay attacks. Failing this allows online bruteforcing of secrets which is not something you want.
- **Whitelisting is your friend and effectively eliminates SQL injection** and other nasty attacks on your precious data. More details on this in the `check_` methods.

- **Require all callback URIs to be registered before use. OAuth providers** are in the unique position of being able to restrict which URIs may be submitted, making validation simple and safe. This registration should be done in your Application management interface.

3. Create your composite endpoint

Each of the endpoints can function independently from each other, however for this example it is easier to consider them as one unit. An example of a pre-configured all-in-one OAuth 1 RFC compliant[1] endpoint is given below:

```
# From the previous section on validators
from my_validator import MyRequestValidator

from oauthlib.oauth1 import WebApplicationServer

validator = MyRequestValidator()
server = WebApplicationServer(validator)
```

Relevant sections include:

Preconfigured all-in-one servers

A pre configured server is an all-in-one endpoint serving a specific class of application clients. As the individual endpoints, they depend on the use of a *Request Validator*.

Construction is simple, only import your validator and you are good to go:

```
from your_validator import your_validator
from oauthlib.oauth1 import WebApplicationServer

server = WebApplicationServer(your_validator)
```

All endpoints are documented in endpoints.

class `oauthlib.oauth1.`**`WebApplicationServer`**(*request_validator*)

4. Create your endpoint views

Standard 3 legged OAuth requires 4 views, request and access token together with pre- and post-authorization. In addition an error view should be defined where users can be informed of invalid/malicious authorization requests.

The example uses Flask but should be transferable to any framework.

```
from flask import Flask, redirect, Response, request, url_for
from oauthlib.oauth1 import OAuth1Error
import urlparse


app = Flask(__name__)


@app.route('/request_token', methods=['POST'])
def request_token():
    h, b, s = provider.create_request_token_response(request.url,
            http_method=request.method,
```

[1] Standard 3-legged OAuth 1 as defined in the RFC specification.

```
            body=request.data,
            headers=request.headers)
    return Response(b, status=s, headers=h)


@app.route('/authorize', methods=['GET'])
def pre_authorize():
    realms, credentials = provider.get_realms_and_credentials(request.url,
            http_method=request.method,
            body=request.data,
            headers=request.headers)
    client_key = credentials.get('resource_owner_key', 'unknown')
    response = '<h1> Authorize access to %s </h1>' % client_key
    response += '<form method="POST" action="/authorize">'
    for realm in realms or []:
        response += ('<input type="checkbox" name="realms" ' +
                         'value="%s"/> %s' % (realm, realm))
    response += '<input type="submit" value="Authorize"/>'
    return response


@app.route('/authorize', methods=['POST'])
def post_authorize():
    realms = request.form.getlist('realms')
    try:
        h, b, s = provider.create_authorization_response(request.url,
                http_method=request.method,
                body=request.data,
                headers=request.headers,
                realms=realms)
        if s == 200:
            return 'Your verifier is: ' + str(urlparse.parse_qs(b)['oauth_verifier
↪'][0])
        else:
            return Response(b, status=s, headers=h)
    except OAuth1Error as e:
        return redirect(e.in_uri(url_for('/error')))


@app.route('/access_token', methods=['POST'])
def access_token():
    h, b, s = provider.create_access_token_response(request.url,
            http_method=request.method,
            body=request.data,
            headers=request.headers)
    return Response(b, status=s, headers=h)


@app.route('/error', methods=['GET'])
def error():
    # Invalid request token will be most likely
    # Could also be an attempt to change the authorization form to try and
    # authorize realms outside the allowed for this client.
    return 'client did something bad'
```

5. Protect your APIs using realms

Let's define a decorator we can use to protect the views.

```
def oauth_protected(realms=None):
    def wrapper(f):
        @functools.wraps(f)
        def verify_oauth(*args, **kwargs):
            validator = OAuthValidator()  # your validator class
            provider = ResourceEndpoint(validator)
            v, r = provider.validate_protected_resource_request(request.url,
                    http_method=request.method,
                    body=request.data,
                    headers=request.headers,
                    realms=realms or [])
            if v:
                return f(*args, **kwargs)
            else:
                return abort(403)
        return verify_oauth
    return wrapper
```

At this point you are ready to protect your API views with OAuth. Take some time to come up with a good set of realms as they can be very powerful in controlling access.

```
@app.route('/secret', methods=['GET'])
@oauth_protected(realms=['secret'])
def protected_resource():
    return 'highly confidential'
```

6. Try your provider with a quick CLI client

This example assumes you use the client key *key* and client secret *secret* shown below as well as run your flask server locally on port *5000*.

```
$ pip install requests requests-oauthlib
```

```
>>> key = 'abcdefghijklmnopqrstuvxyzabcde'
>>> secret = 'foo'

>>> # OAuth endpoints given in the Bitbucket API documentation
>>> request_token_url = 'http://127.0.0.1:5000/request_token'
>>> authorization_base_url = 'http://127.0.0.1:5000/authorize'
>>> access_token_url = 'http://127.0.0.1:5000/access_token'

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

>>> # 3. Redirect user to your provider implementation for authorization
>>> authorization_url = oauth.authorization_url(authorization_base_url)
>>> print 'Please go here and authorize,', authorization_url

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

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

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

>>> # 6. Fetch a protected resource, i.e. user profile
>>> r = oauth.get('http://127.0.0.1:5000/secret')
>>> print r.content
```

7. Let us know how it went!

Drop a line in our G+ community or open a GitHub issue =)

If you run into issues it can be helpful to enable debug logging:

```
import logging
import sys
log = logging.getLogger('oauthlib')
log.addHandler(logging.StreamHandler(sys.stdout))
log.setLevel(logging.DEBUG)
```

Provider endpoints

Each endpoint is responsible for one step in the OAuth 1 workflow. They can be used either independently or in a combination. They depend on the use of a *Request Validator*.

See *Preconfigured all-in-one servers* for available composite endpoints/servers.

Request Token

class `oauthlib.oauth1.`**`RequestTokenEndpoint`**(*request_validator*, *token_generator=None*)

An endpoint responsible for providing OAuth 1 request tokens.

Typical use is to instantiate with a request validator and invoke the `create_request_token_response` from a view function. The tuple returned has all information necessary (body, status, headers) to quickly form and return a proper response. See *Request Validator* for details on which validator methods to implement for this endpoint.

`create_request_token`(*request*, *credentials*)

Create and save a new request token.

Parameters

- **`request`** – An oauthlib.common.Request object.
- **`credentials`** – A dict of extra token credentials.

Returns The token as an urlencoded string.

`create_request_token_response`(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*, *credentials=None*)

Create a request token response, with a new request token if valid.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.
- **credentials** – A list of extra credentials to include in the token.

Returns A tuple of 3 elements. 1. A dict of headers to set on the response. 2. The response body as a string. 3. The response status code as an integer.

An example of a valid request:

```
>>> from your_validator import your_validator
>>> from oauthlib.oauth1 import RequestTokenEndpoint
>>> endpoint = RequestTokenEndpoint(your_validator)
>>> h, b, s = endpoint.create_request_token_response(
...     'https://your.provider/request_token?foo=bar',
...     headers={
...         'Authorization': 'OAuth realm=movies user, oauth_....'
...     },
...     credentials={
...         'my_specific': 'argument',
...     })
>>> h
{'Content-Type': 'application/x-www-form-urlencoded'}
>>> b
'oauth_token=lsdkfol23w54jlksdef&oauth_token_secret=qwe089234lkjsdf&oauth_
↪callback_confirmed=true&my_specific=argument'
>>> s
200
```

An response to invalid request would have a different body and status:

```
>>> b
'error=invalid_request&description=missing+callback+uri'
>>> s
400
```

The same goes for an an unauthorized request:

```
>>> b
''
>>> s
401
```

validate_request_token_request(*request*)
Validate a request token request.

Parameters **request** – An oauthlib.common.Request object.

Raises OAuth1Error if the request is invalid.

Returns A tuple of 2 elements. 1. The validation result (True or False). 2. The request object.

Authorization

class oauthlib.oauth1.**AuthorizationEndpoint**(*request_validator*, *token_generator=None*)
An endpoint responsible for letting authenticated users authorize access to their protected resources to a client.

Typical use would be to have two views, one for displaying the authorization form and one to process said form on submission.

The first view will want to utilize `get_realms_and_credentials` to fetch requested realms and useful client credentials, such as name and description, to be used when creating the authorization form.

During form processing you can use `create_authorization_response` to validate the request, create a verifier as well as prepare the final redirection URI used to send the user back to the client.

See *Request Validator* for details on which validator methods to implement for this endpoint.

create_authorization_response(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*, *realms=None*, *credentials=None*)

Create an authorization response, with a new request token if valid.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.
- **credentials** – A list of credentials to include in the verifier.

Returns A tuple of 3 elements. 1. A dict of headers to set on the response. 2. The response body as a string. 3. The response status code as an integer.

If the callback URI tied to the current token is "oob", a response with a 200 status code will be returned. In this case, it may be desirable to modify the response to better display the verifier to the client.

An example of an authorization request:

```
>>> from your_validator import your_validator
>>> from oauthlib.oauth1 import AuthorizationEndpoint
>>> endpoint = AuthorizationEndpoint(your_validator)
>>> h, b, s = endpoint.create_authorization_response(
...     'https://your.provider/authorize?oauth_token=...',
...     credentials={
...         'extra': 'argument',
...     })
>>> h
{'Location': 'https://the.client/callback?oauth_verifier=...&extra=argument'}
>>> b
None
>>> s
302
```

An example of a request with an "oob" callback:

```
>>> from your_validator import your_validator
>>> from oauthlib.oauth1 import AuthorizationEndpoint
>>> endpoint = AuthorizationEndpoint(your_validator)
>>> h, b, s = endpoint.create_authorization_response(
...     'https://your.provider/authorize?foo=bar',
...     credentials={
...         'extra': 'argument',
...     })
>>> h
{'Content-Type': 'application/x-www-form-urlencoded'}
>>> b
```

```
'oauth_verifier=...&extra=argument'
>>> s
200
```

create_verifier(*request*, *credentials*)
Create and save a new request token.

Parameters

- **request** – An oauthlib.common.Request object.
- **credentials** – A dict of extra token credentials.

Returns The verifier as a dict.

get_realms_and_credentials(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*)
Fetch realms and credentials for the presented request token.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.

Returns A tuple of 2 elements. 1. A list of request realms. 2. A dict of credentials which may be useful in creating the authorization form.

Access Token

class oauthlib.oauth1.**AccessTokenEndpoint**(*request_validator*, *token_generator=None*)
An endpoint responsible for providing OAuth 1 access tokens.

Typical use is to instantiate with a request validator and invoke the `create_access_token_response` from a view function. The tuple returned has all information necessary (body, status, headers) to quickly form and return a proper response. See *Request Validator* for details on which validator methods to implement for this endpoint.

create_access_token(*request*, *credentials*)
Create and save a new access token.

Similar to OAuth 2, indication of granted scopes will be included as a space separated list in `oauth_authorized_realms`.

Parameters **request** – An oauthlib.common.Request object.

Returns The token as an urlencoded string.

create_access_token_response(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*, *credentials=None*)
Create an access token response, with a new request token if valid.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.

- **credentials** – A list of extra credentials to include in the token.

Returns A tuple of 3 elements. 1. A dict of headers to set on the response. 2. The response body as a string. 3. The response status code as an integer.

An example of a valid request:

```
>>> from your_validator import your_validator
>>> from oauthlib.oauth1 import AccessTokenEndpoint
>>> endpoint = AccessTokenEndpoint(your_validator)
>>> h, b, s = endpoint.create_access_token_response(
...     'https://your.provider/access_token?foo=bar',
...     headers={
...         'Authorization': 'OAuth oauth_token=234lsdkf....'
...     },
...     credentials={
...         'my_specific': 'argument',
...     })
>>> h
{'Content-Type': 'application/x-www-form-urlencoded'}
>>> b
'oauth_token=lsdkfol23w54jlksdef&oauth_token_secret=qwe089234lkjsdf&oauth_
↪authorized_realms=movies+pics&my_specific=argument'
>>> s
200
```

An response to invalid request would have a different body and status:

```
>>> b
'error=invalid_request&description=missing+resource+owner+key'
>>> s
400
```

The same goes for an an unauthorized request:

```
>>> b
''
>>> s
401
```

validate_access_token_request(*request*)

Validate an access token request.

Parameters **request** – An oauthlib.common.Request object.

Raises OAuth1Error if the request is invalid.

Returns A tuple of 2 elements. 1. The validation result (True or False). 2. The request object.

Resource authorization

class oauthlib.oauth1.**ResourceEndpoint**(*request_validator*, *token_generator=None*)

An endpoint responsible for protecting resources.

Typical use is to instantiate with a request validator and invoke the `validate_protected_resource_request` in a decorator around a view function. If the request is valid, invoke and return the response of the view. If invalid create and return an error response directly from the decorator.

See *Request Validator* for details on which validator methods to implement for this endpoint.

An example decorator:

```
from functools import wraps
from your_validator import your_validator
from oauthlib.oauth1 import ResourceEndpoint
endpoint = ResourceEndpoint(your_validator)

def require_oauth(realms=None):
    def decorator(f):
        @wraps(f)
        def wrapper(request, *args, **kwargs):
            v, r = provider.validate_protected_resource_request(
                    request.url,
                    http_method=request.method,
                    body=request.data,
                    headers=request.headers,
                    realms=realms or [])
            if v:
                return f(*args, **kwargs)
            else:
                return abort(403)
```

validate_protected_resource_request(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*, *realms=None*)

Create a request token response, with a new request token if valid.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.
- **realms** – A list of realms the resource is protected under. This will be supplied to the `validate_realms` method of the request validator.

Returns A tuple of 2 elements. 1. True if valid, False otherwise. 2. An oauthlib.common.Request object.

Signature Only

class oauthlib.oauth1.**SignatureOnlyEndpoint**(*request_validator*, *token_generator=None*)

An endpoint only responsible for verifying an oauth signature.

validate_request(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*)

Validate a signed OAuth request.

Parameters

- **uri** – The full URI of the token request.
- **http_method** – A valid HTTP verb, i.e. GET, POST, PUT, HEAD, etc.
- **body** – The request body as a string.
- **headers** – The request headers as a dict.

Returns A tuple of 2 elements. 1. True if valid, False otherwise. 2. An oauthlib.common.Request object.

CHAPTER 9

OAuth 2.0

Security

OAuth 2 is much simpler to implement for clients than OAuth 1 as cryptographic signing is no longer necessary. Instead a strict requirement on the use of TLS for all connections have been introduced.

> **Warning:** OAuthLib will raise errors if you attempt to interact with a non HTTPS endpoint during authorization. However OAuthLib offers no such protection during token requests as the URI is not provided, only the request body.

Note that while OAuth 2 is simpler it does subtly transfer a few important responsibilities from the provider to the client. Most notably that the client must ensure that all tokens are kept secret at all times. Access to protected resources using Bearer tokens provides no authenticity of clients which means that a malicious party able to obtain your tokens can use them without the provider being able to know the difference. This is unlike OAuth 1 where a lost token could not be utilized without the client secret and the token bound secret, since they are required for the signing of each request.

Environment Variables

It is possible to customize some of the security settings in OAuthLib using environment variables. You can use this to bypass some of OAuthLib's security checks in order to run automated tests. *Never* bypass these checks in production.

OAUTHLIB_INSECURE_TRANSPORT
: Normally, OAuthLib will raise an `InsecureTransportError` if you attempt to use OAuth2 over HTTP, rather than HTTPS. Setting this environment variable will prevent this error from being raised. This is mostly useful for local testing, or automated tests. *Never* set this variable in production.

OAUTHLIB_STRICT_TOKEN_TYPE
: When parsing an OAuth2 token response, OAuthLib normally ignores the `token_type` parameter. Set-

ting this variable will cause OAuthLib to specifically check for this parameter in the response, and raise an `MissingTokenTypeError` if the parameter is missing.

Using Clients

OAuthLib supports all four core grant types defined in the OAuth 2 RFC and will continue to add more as they are defined. For more information on how to use them please browse the documentation for each client type below.

Base Client

class `oauthlib.oauth2.`**`Client`**(*client_id*, *default_token_placement=u'auth_header'*, *token_type=u'Bearer'*, *access_token=None*, *refresh_token=None*, *mac_key=None*, *mac_algorithm=None*, *token=None*, *scope=None*, *state=None*, *redirect_url=None*, *state_generator=<function generate_token>*, ***kwargs*)

Base OAuth2 client responsible for access token management.

This class also acts as a generic interface providing methods common to all client types such as `prepare_authorization_request` and `prepare_token_revocation_request`. The `prepare_x_request` methods are the recommended way of interacting with clients (as opposed to the abstract prepare uri/body/etc methods). They are recommended over the older set because they are easier to use (more consistent) and add a few additional security checks, such as HTTPS and state checking.

Some of these methods require further implementation only provided by the specific purpose clients such as *`oauthlib.oauth2.MobileApplicationClient`* and thus you should always seek to use the client class matching the OAuth workflow you need. For Python, this is usually *`oauthlib.oauth2.WebApplicationClient`*.

`add_token`(*uri*, *http_method=u'GET'*, *body=None*, *headers=None*, *token_placement=None*, ***kwargs*)

Add token to the request uri, body or authorization header.

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client MUST NOT use an access token if it does not understand the token type.

For example, the "bearer" token type defined in [I-D.ietf-oauth-v2-bearer] is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

while the "mac" token type defined in [I-D.ietf-oauth-v2-http-mac] is utilized by issuing a MAC key together with the access token which is used to sign certain components of the HTTP requests:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC id="h480djs93hd8",
                   nonce="274312:dj83hs9s",
                   mac="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

`parse_request_body_response`(*body*, *scope=None*, ***kwargs*)

Parse the JSON response body.

If the access token request is valid and authorized, the authorization server issues an access token as described in Section 5.1. A refresh token SHOULD NOT be included. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

Parameters

- **body** – The response body from the token request.
- **scope** – Scopes originally requested.

Returns Dictionary of token parameters.

Raises Warning if scope has changed. OAuth2Error if response is invalid.

These response are json encoded and could easily be parsed without the assistance of OAuthLib. However, there are a few subtle issues to be aware of regarding the response which are helpfully addressed through the raising of various errors.

A successful response should always contain

access_token The access token issued by the authorization server. Often a random string.

token_type The type of the token issued as described in Section 7.1. Commonly `Bearer`.

While it is not mandated it is recommended that the provider include

expires_in The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

scope Providers may supply this in all responses but are required to only if it has changed since the authorization request.

parse_request_uri_response(**args*, ***kwargs*)
Abstract method used to parse redirection responses.

prepare_authorization_request(*authorization_url*, *state=None*, *redirect_url=None*, *scope=None*, ***kwargs*)
Prepare the authorization request.

This is the first step in many OAuth flows in which the user is redirected to a certain authorization URL. This method adds required parameters to the authorization URL.

Parameters

- **authorization_url** – Provider authorization endpoint URL.
- **state** – CSRF protection string. Will be automatically created if

not provided. The generated state is available via the `state` attribute. Clients should verify that the state is unchanged and present in the authorization response. This verification is done automatically if using the `authorization_response` parameter with `prepare_token_request`.

Parameters **redirect_url** – Redirect URL to which the user will be returned

after authorization. Must be provided unless previously setup with the provider. If provided then it must also be provided in the token request.

Parameters **kwargs** – Additional parameters to included in the request.

Returns The prepared request tuple with (url, headers, body).

prepare_refresh_body(*body=u''*, *refresh_token=None*, *scope=None*, ***kwargs*)
Prepare an access token request, using a refresh token.

If the authorization server issued a refresh token to the client, the client makes a refresh request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

grant_type REQUIRED. Value MUST be set to "refresh_token".

refresh_token REQUIRED. The refresh token issued to the client.

scope OPTIONAL. The scope of the access request as described by Section 3.3. The requested scope MUST NOT include any scope not originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted by the resource owner.

prepare_refresh_token_request(*token_url*, *refresh_token=None*, *body=u''*, *scope=None*, ***kwargs*)

Prepare an access token refresh request.

Expired access tokens can be replaced by new access tokens without going through the OAuth dance if the client obtained a refresh token. This refresh token and authentication credentials can be used to obtain a new access token, and possibly a new refresh token.

Parameters

- **token_url** – Provider token refresh endpoint URL.
- **refresh_token** – Refresh token string.
- **body** – Request body (URL encoded string).
- **scope** – List of scopes to request. Must be equal to

or a subset of the scopes granted when obtaining the refresh token.

Parameters **kwargs** – Additional parameters to included in the request.

Returns The prepared request tuple with (url, headers, body).

prepare_request_body(**args*, ***kwargs*)

Abstract method used to create request bodies.

prepare_request_uri(**args*, ***kwargs*)

Abstract method used to create request URIs.

prepare_token_request(*token_url*, *authorization_response=None*, *redirect_url=None*, *state=None*, *body=u''*, ***kwargs*)

Prepare a token creation request.

Note that these requests usually require client authentication, either by including client_id or a set of provider specific authentication credentials.

Parameters

- **token_url** – Provider token creation endpoint URL.
- **authorization_response** – The full redirection URL string, i.e.

the location to which the user was redirected after successfull authorization. Used to mine credentials needed to obtain a token in this step, such as authorization code.

Parameters **redirect_url** – The redirect_url supplied with the authorization

request (if there was one).

Parameters

- **body** – Request body (URL encoded string).
- **kwargs** – Additional parameters to included in the request.

Returns The prepared request tuple with (url, headers, body).

prepare_token_revocation_request(*revocation_url*, *token*, *token_type_hint=u'access_token'*, *body=u''*, *callback=None*, ***kwargs*)

Prepare a token revocation request.

Parameters

- **revocation_url** – Provider token revocation endpoint URL.
- **token** – The access or refresh token to be revoked (string).
- **token_type_hint** – `"access_token"` (default) or

`"refresh_token"`. This is optional and if you wish to not pass it you must provide `token_type_hint=None`.

Parameters **callback** – A jsonp callback such as `package.callback` to be invoked

upon receiving the response. Not that it should not include a () suffix.

Parameters **kwargs** – Additional parameters to included in the request.

Returns The prepared request tuple with (url, headers, body).

Note that JSONP request may use GET requests as the parameters will be added to the request URL query as opposed to the request body.

An example of a revocation request

An example of a jsonp revocation request

and an error response

package.myCallback({"error":"unsupported_token_type"});

Note that these requests usually require client credentials, client_id in the case for public clients and provider specific authentication credentials for confidential clients.

token_types

Supported token types and their respective methods

Additional tokens can be supported by extending this dictionary.

The Bearer token spec is stable and safe to use.

The MAC token spec is not yet stable and support for MAC tokens is experimental and currently matching version 00 of the spec.

WebApplicationClient

class oauthlib.oauth2.**WebApplicationClient**(*client_id*, *code=None*, ***kwargs*)

A client utilizing the authorization code grant workflow.

A web application is a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent on the device used by the resource owner. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

parse_request_uri_response(*uri*, *state=None*)
Parse the URI query for code and state.

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format:

Parameters

- **uri** – The callback URI that resulted from the user being redirected back from the provider to you, the client.
- **state** – The state provided in the authorization request.

code The authorization code generated by the authorization server. The authorization code MUST expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The client MUST NOT use the authorization code more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state If the "state" parameter was present in the authorization request.

This method is mainly intended to enforce strict state checking with the added benefit of easily extracting parameters from the URI:

```
>>> from oauthlib.oauth2 import WebApplicationClient
>>> client = WebApplicationClient('your_id')
>>> uri = 'https://example.com/callback?code=sdfkjh345&state=sfetw45'
>>> client.parse_request_uri_response(uri, state='sfetw45')
{'state': 'sfetw45', 'code': 'sdfkjh345'}
>>> client.parse_request_uri_response(uri, state='other')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "oauthlib/oauth2/rfc6749/__init__.py", line 357, in parse_request_
↪uri_response
        back from the provider to you, the client.
    File "oauthlib/oauth2/rfc6749/parameters.py", line 153, in parse_
↪authorization_code_response
        raise MismatchingStateError()
oauthlib.oauth2.rfc6749.errors.MismatchingStateError
```

prepare_request_body(*client_id=None*, *code=None*, *body=u''*, *redirect_uri=None*, ***kwargs*)
Prepare the access token request body.

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

Parameters

- **client_id** – REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.
- **code** – REQUIRED. The authorization code received from the authorization server.
- **redirect_uri** – REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.
- **kwargs** – Extra parameters to include in the token request.

In addition OAuthLib will add the `grant_type` parameter set to `authorization_code`.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1:

```
>>> from oauthlib.oauth2 import WebApplicationClient
>>> client = WebApplicationClient('your_id')
>>> client.prepare_request_body(code='sh35ksdf09sf')
'grant_type=authorization_code&code=sh35ksdf09sf'
>>> client.prepare_request_body(code='sh35ksdf09sf', foo='bar')
'grant_type=authorization_code&code=sh35ksdf09sf&foo=bar'
```

prepare_request_uri(*uri*, *redirect_uri=None*, *scope=None*, *state=None*, ***kwargs*)

Prepare the authorization code request URI

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format, per Appendix B:

Parameters

- **redirect_uri** – OPTIONAL. The redirect URI must be an absolute URI and it should have been registerd with the OAuth provider prior to use. As described in Section 3.1.2.
- **scope** – OPTIONAL. The scope of the access request as described by Section 3.3'_. These may be any string but are commonly URIs or various categories such as `videos` or `documents`.
- **state** – RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.
- **kwargs** – Extra arguments to include in the request URI.

In addition to supplied parameters, OAuthLib will append the `client_id` that was provided in the constructor as well as the mandatory `response_type` argument, set to `code`:

```
>>> from oauthlib.oauth2 import WebApplicationClient
>>> client = WebApplicationClient('your_id')
>>> client.prepare_request_uri('https://example.com')
'https://example.com?client_id=your_id&response_type=code'
>>> client.prepare_request_uri('https://example.com', redirect_uri='https://a.
↪b/callback')
'https://example.com?client_id=your_id&response_type=code&redirect_uri=https
↪%3A%2F%2Fa.b%2Fcallback'
>>> client.prepare_request_uri('https://example.com', scope=['profile',
↪'pictures'])
'https://example.com?client_id=your_id&response_type=code&
↪scope=profile+pictures'
>>> client.prepare_request_uri('https://example.com', foo='bar')
'https://example.com?client_id=your_id&response_type=code&foo=bar'
```

MobileApplicationClient

class oauthlib.oauth2.**MobileApplicationClient**(*client_id*, *default_token_placement=u'auth_header'*, *token_type=u'Bearer'*, *access_token=None*, *refresh_token=None*, *mac_key=None*, *mac_algorithm=None*, *token=None*, *scope=None*, *state=None*, *redirect_url=None*, *state_generator=<function generate_token>*, ***kwargs*)

A public client utilizing the implicit code grant workflow.

A user-agent-based application is a public client in which the client code is downloaded from a web server and executes within a user-agent (e.g. web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner. Since such applications reside within the user-agent, they can make seamless use of the user-agent capabilities when requesting authorization.

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

Unlike the authorization code grant type in which the client makes separate requests for authorization and access token, the client receives the access token as the result of the authorization request.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device.

parse_request_uri_response(*uri*, *state=None*, *scope=None*)

Parse the response URI fragment.

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format:

Parameters

- **uri** – The callback URI that resulted from the user being redirected back from the provider to you, the client.
- **state** – The state provided in the authorization request.
- **scope** – The scopes provided in the authorization request.

Returns Dictionary of token parameters.

Raises OAuth2Error if response is invalid.

A successful response should always contain

access_token The access token issued by the authorization server. Often a random string.

token_type The type of the token issued as described in Section 7.1. Commonly `Bearer`.

state If you provided the state parameter in the authorization phase, then the provider is required to include that exact state value in the response.

While it is not mandated it is recommended that the provider include

expires_in The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

scope Providers may supply this in all responses but are required to only if it has changed since the authorization request.

A few example responses can be seen below:

```
>>> response_uri = 'https://example.com/callback#access_token=sdlfkj452&
↪state=ss345asyht&token_type=Bearer&scope=hello+world'
>>> from oauthlib.oauth2 import MobileApplicationClient
>>> client = MobileApplicationClient('your_id')
>>> client.parse_request_uri_response(response_uri)
{
    'access_token': 'sdlfkj452',
    'token_type': 'Bearer',
    'state': 'ss345asyht',
    'scope': [u'hello', u'world']
}
>>> client.parse_request_uri_response(response_uri, state='other')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "oauthlib/oauth2/rfc6749/__init__.py", line 598, in parse_request_
↪uri_response
        **scope**
    File "oauthlib/oauth2/rfc6749/parameters.py", line 197, in parse_implicit_
↪response
        raise ValueError("Mismatching or missing state in params.")
ValueError: Mismatching or missing state in params.
>>> def alert_scope_changed(message, old, new):
...     print(message, old, new)
...
>>> oauthlib.signals.scope_changed.connect(alert_scope_changed)
>>> client.parse_request_body_response(response_body, scope=['other'])
('Scope has changed from "other" to "hello world".', ['other'], ['hello',
↪'world'])
```

prepare_request_uri(*uri*, *redirect_uri=None*, *scope=None*, *state=None*, ***kwargs*)

Prepare the implicit grant request URI.

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format, per Appendix B:

Parameters

- **redirect_uri** – OPTIONAL. The redirect URI must be an absolute URI and it should have been registerd with the OAuth provider prior to use. As described in Section 3.1.2.
- **scope** – OPTIONAL. The scope of the access request as described by Section 3.3'_. These may be any string but are commonly URIs or various categories such as `videos` or `documents`.
- **state** – RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.
- **kwargs** – Extra arguments to include in the request URI.

In addition to supplied parameters, OAuthLib will append the `client_id` that was provided in the constructor as well as the mandatory `response_type` argument, set to `token`:

```
>>> from oauthlib.oauth2 import MobileApplicationClient
>>> client = MobileApplicationClient('your_id')
>>> client.prepare_request_uri('https://example.com')
'https://example.com?client_id=your_id&response_type=token'
>>> client.prepare_request_uri('https://example.com', redirect_uri='https://a.
↪b/callback')
'https://example.com?client_id=your_id&response_type=token&redirect_uri=https
↪%3A%2F%2Fa.b%2Fcallback'
>>> client.prepare_request_uri('https://example.com', scope=['profile',
↪'pictures'])
'https://example.com?client_id=your_id&response_type=token&
↪scope=profile+pictures'
>>> client.prepare_request_uri('https://example.com', foo='bar')
'https://example.com?client_id=your_id&response_type=token&foo=bar'
```

LegacyApplicationClient

class `oauthlib.oauth2.`**LegacyApplicationClient**(*client_id*, ***kwargs*)

A public client using the resource owner password and username directly.

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. The authorization server should take special care when enabling this grant type, and only allow it when other flows are not viable.

The grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

The method through which the client obtains the resource owner credentials is beyond the scope of this specification. The client MUST discard the credentials once an access token has been obtained.

prepare_request_body(*username*, *password*, *body=u''*, *scope=None*, ***kwargs*)

Add the resource owner password and username to the request body.

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B in the HTTP request entity-body:

Parameters

- **username** – The resource owner username.
- **password** – The resource owner password.
- **scope** – The scope of the access request as described by Section 3.3.
- **kwargs** – Extra credentials to include in the token request.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

The prepared body will include all provided credentials as well as the `grant_type` parameter set to `password`:

```
>>> from oauthlib.oauth2 import LegacyApplicationClient
>>> client = LegacyApplicationClient('your_id')
```

```
>>> client.prepare_request_body(username='foo', password='bar', scope=['hello
↪', 'world'])
'grant_type=password&username=foo&scope=hello+world&password=bar'
```

BackendApplicationClient

class oauthlib.oauth2.**BackendApplicationClient**(*client_id*, *default_token_placement=u'auth_header'*, *token_type=u'Bearer'*, *access_token=None*, *refresh_token=None*, *mac_key=None*, *mac_algorithm=None*, *token=None*, *scope=None*, *state=None*, *redirect_url=None*, *state_generator=<function generate_token>*, ***kwargs*)

A public client utilizing the client credentials grant workflow.

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner which has been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients.

Since the client authentication is used as the authorization grant, no additional authorization request is needed.

prepare_request_body(*body=u''*, *scope=None*, ***kwargs*)

Add the client credentials to the request body.

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B in the HTTP request entity-body:

Parameters

- **scope** – The scope of the access request as described by Section 3.3.
- **kwargs** – Extra credentials to include in the token request.

The client MUST authenticate with the authorization server as described in Section 3.2.1.

The prepared body will include all provided credentials as well as the `grant_type` parameter set to `client_credentials`:

```
>>> from oauthlib.oauth2 import BackendApplicationClient
>>> client = BackendApplicationClient('your_id')
>>> client.prepare_request_body(scope=['hello', 'world'])
'grant_type=client_credentials&scope=hello+world'
```

Existing libraries If you are using the requests HTTP library you may be interested in using requests-oauthlib which provides an OAuth 2 Client. This client removes much of the boilerplate you might otherwise need to deal with when interacting with OAuthLib directly.

If you are interested in integrating OAuth 2 support into your favourite HTTP library you might find the requests-oauthlib implementation interesting.

Creating a Provider

OAuthLib is a dependency free library that may be used with any web framework. That said, there are framework specific helper libraries to make your life easier.

- For Django there is django-oauth-toolkit.
- For Flask there is flask-oauthlib.

If there is no support for your favourite framework and you are interested in providing it then you have come to the right place. OAuthLib can handle the OAuth logic and leave you to support a few framework and setup specific tasks such as marshalling request objects into URI, headers and body arguments as well as provide an interface for a backend to store tokens, clients, etc.

Tutorial Contents

1. Create your datastore models

These models will represent various OAuth specific concepts. There are a few important links between them that the security of OAuth is based on. Below is a suggestion for models and why you need certain properties. There is also example Django model fields which should be straightforward to translate to other ORMs such as SQLAlchemy and the Appengine Datastore.

User (or Resource Owner)

The user of your site which resources might be accessed by clients upon authorization from the user. In our example we will re-use the User model provided in django.contrib.auth.models. How the user authenticates is orthogonal from OAuth and may be any way you prefer:

```
from django.contrib.auth.models import User
```

Client (or Consumer)

The client interested in accessing protected resources.

Client Identifier:

Required. The identifier the client will use during the OAuth workflow. Structure is up to you and may be a simple UUID.

```
client_id = django.db.models.CharField(max_length=100, unique=True)
```

User:

Recommended. It is common practice to link each client with one of your existing users. Whether you do associate clients and users or not, ensure you are able to protect yourself against malicious clients.

```
user = django.db.models.ForeignKey(User)
```

Grant Type:

Required. The grant type the client may utilize. This should only be one per client as each grant type has different security properties and it is best to keep them separate to avoid mistakes.

```
# max_length and choices depend on which response types you support
grant_type = django.db.models.CharField(max_length=18,
choices=[('authorization_code', 'Authorization code')])
```

Response Type:

Required, if using a grant type with an associated response type (eg. Authorization Code Grant) or using a grant which only utilizes response types (eg. Implicit Grant).

```
# max_length and choices depend on which response types you support
response_type = django.db.models.CharField(max_length=4,
choices=[('code', 'Authorization code')])
```

Scopes:

Required. The list of scopes the client may request access to. If you allow multiple types of grants this will vary related to their different security properties. For example, the Implicit Grant might only allow read-only scopes but the Authorization Grant also allow writes.

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
scopes = django.db.models.TextField()

# You might also want to mark a certain set of scopes as default
# scopes in case the client does not specify any in the authorization
default_scopes = django.db.models.TextField()
```

Redirect URIs:

These are the absolute URIs that a client may use to redirect to after authorization. You should never allow a client to redirect to a URI that has not previously been registered.

```
# You could represent the URIs either as a list of keys or by
# serializing them into a string.
redirect_uris = django.db.models.TextField()

# You might also want to mark a certain URI as default in case the
# client does not specify any in the authorization
default_redirect_uri = django.db.models.TextField()
```

Bearer Token (OAuth 2 Standard Token)

The most common type of OAuth 2 token. Through the documentation this will be considered an object with several properties, such as token type and expiration date, and distinct from the access token it contains. Think of OAuth 2 tokens as containers and access tokens and refresh tokens as text.

Client:

Association with the client to whom the token was given.

```
client = django.db.models.ForeignKey(Client)
```

User:

Association with the user to which protected resources this token grants access.

```
user = django.db.models.ForeignKey(User)
```

Scopes:

Scopes to which the token is bound. Attempt to access protected resources outside these scopes will be denied.

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
scopes = django.db.models.TextField()
```

Access Token:

An unguessable unique string of characters.

```
access_token = django.db.models.CharField(max_length=100, unique=True)
```

Refresh Token:

An unguessable unique string of characters. This token is only supplied to confidential clients. For example the Authorization Code Grant or the Resource Owner Password Credentials Grant.

```
refresh_token = django.db.models.CharField(max_length=100, unique=True)
```

Expiration time:

Exact time of expiration. Commonly this is one hour after creation.

```
expires_at = django.db.models.DateTimeField()
```

Authorization Code

This is specific to the Authorization Code grant and represent the temporary credential granted to the client upon successful authorization. It will later be exchanged for an access token, when that is done it should cease to exist. It should have a limited life time, less than ten minutes. This model is similar to the Bearer Token as it mainly acts a temporary storage of properties to later be transferred to the token.

Client:

Association with the client to whom the token was given.

```
client = django.db.models.ForeignKey(Client)
```

User:

Association with the user to which protected resources this token grants access.

```
user = django.db.models.ForeignKey(User)
```

Scopes:

Scopes to which the token is bound. Attempt to access protected resources outside these scopes will be denied.

```
# You could represent it either as a list of keys or by serializing
# the scopes into a string.
scopes = django.db.models.TextField()
```

Authorization Code:

An unguessable unique string of characters.

```
code = django.db.models.CharField(max_length=100, unique=True)
```

Expiration time:

Exact time of expiration. Commonly this is under ten minutes after creation.

```
expires_at = django.db.models.DateTimeField()
```

2. Implement a validator

The majority of the work involved in implementing an OAuth 2 provider relates to mapping various validation and persistence methods to a storage backend. The not very accurately named interface you will need to implement is called a *RequestValidator* (name suggestions welcome).

An example of a very basic implementation of the validate_client_id method can be seen below.

```
from oauthlib.oauth2 import RequestValidator

# From the previous section on models
from my_models import Client

class MyRequestValidator(RequestValidator):

    def validate_client_id(self, client_id, request):
        try:
            Client.objects.get(client_id=client_id)
            return True
        except Client.DoesNotExist:
            return False
```

The full API you will need to implement is available in the *RequestValidator* section. You might not need to implement all methods depending on which grant types you wish to support. A skeleton validator listing the methods required for the WebApplicationServer is available in the examples folder on GitHub.

Relevant sections include:

Request Validator

class oauthlib.oauth2.**RequestValidator**

authenticate_client(*request*, **args*, ***kwargs*)
Authenticate client through means outside the OAuth 2 spec.

Means of authentication is negotiated beforehand and may for example be HTTP Basic Authentication Scheme which utilizes the Authorization header.

Headers may be accesses through request.headers and parameters found in both body and query can be obtained by direct attribute access, i.e. request.client_id for client_id in the URL query.

Parameters **request** – oauthlib.common.Request

Return type True or False

Method is used by:

- Authorization Code Grant
- Resource Owner Password Credentials Grant (may be disabled)
- Client Credentials Grant
- Refresh Token Grant

authenticate_client_id(*client_id*, *request*, **args*, ***kwargs*)
Ensure client_id belong to a non-confidential client.

A non-confidential client is one that is not required to authenticate through other means, such as using HTTP Basic.

Note, while not strictly necessary it can often be very convenient to set request.client to the client object associated with the given client_id.

Parameters **request** – oauthlib.common.Request

Return type True or False

Method is used by:

- Authorization Code Grant

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, when Client type is Confidential or when Client was issued client credentials or whenever Client provided client authentication, see Section 4.3.2.
- Authorization Code Grant, when Client type is Confidential or when Client was issued client credentials or whenever Client provided client authentication, see Section 4.1.3.
- Refresh Token Grant, when Client type is Confidential or when Client was issued client credentials or whenever Client provided client authentication, see Section 6

Parameters **request** – oauthlib.common.Request

Return type True or False

Method is used by:

- Authorization Code Grant
- Resource Owner Password Credentials Grant
- Refresh Token Grant

confirm_redirect_uri (*client_id*, *code*, *redirect_uri*, *client*, **args*, ***kwargs*)
Ensure that the authorization process represented by this authorization code began with this 'redirect_uri'.

If the client specifies a redirect_uri when obtaining code then that redirect URI must be bound to the code and verified equal in this method, according to RFC 6749 section 4.1.3. Do not compare against the client's allowed redirect URIs, but against the URI used when the code was saved.

Parameters

- **client_id** – Unicode client identifier
- **code** – Unicode authorization_code.
- **redirect_uri** – Unicode absolute URI
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant (during token request)

get_default_redirect_uri (*client_id*, *request*, **args*, ***kwargs*)
Get the default redirect URI for the client.

Parameters

- **client_id** – Unicode client identifier
- **request** – The HTTP Request (oauthlib.common.Request)

Return type The default redirect URI for the client

Method is used by:

- Authorization Code Grant
- Implicit Grant

get_default_scopes (*client_id*, *request*, **args*, ***kwargs*)
Get the default scopes for the client.

Parameters

- **client_id** – Unicode client identifier
- **request** – The HTTP Request (oauthlib.common.Request)

Return type List of default scopes

Method is used by all core grant types:

- Authorization Code Grant
- Implicit Grant

- Resource Owner Password Credentials Grant
- Client Credentials grant

get_id_token(*token*, *token_handler*, *request*)
In the OpenID Connect workflows when an ID Token is requested this method is called. Subclasses should implement the construction, signing and optional encryption of the ID Token as described in the OpenID Connect spec.

In addition to the standard OAuth2 request properties, the request may also contain these OIDC specific properties which are useful to this method:

•nonce, if workflow is implicit or hybrid and it was provided

•claims, if provided to the original Authorization Code request

The token parameter is a dict which may contain an `access_token` entry, in which case the resulting ID Token *should* include a calculated `at_hash` claim.

Similarly, when the request parameter has a `code` property defined, the ID Token *should* include a calculated `c_hash` claim.

http://openid.net/specs/openid-connect-core-1_0.html (sections 3.1.3.6, 3.2.2.10, 3.3.2.11)

Parameters

- **token** – A Bearer token dict
- **token_handler** – the token handler (BearerToken class)
- **request** – the HTTP Request (oauthlib.common.Request)

Returns The ID Token (a JWS signed JWT)

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

Parameters

- **refresh_token** – Unicode refresh token
- **request** – The HTTP Request (oauthlib.common.Request)

Return type List of scopes.

Method is used by:

- Refresh token grant

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

Parameters

- **client_id** – Unicode client identifier
- **code** – The authorization code grant (request.code).
- **request** – The HTTP Request (oauthlib.common.Request)

Method is used by:

- Authorization Code Grant

is_within_original_scope(*request_scopes*, *refresh_token*, *request*, **args*, ***kwargs*)
Check if requested scopes are within a scope of the refresh token.

When access tokens are refreshed the scope of the new token needs to be within the scope of the original token. This is ensured by checking that all requested scopes strings are on the list returned by the get_original_scopes. If this check fails, is_within_original_scope is called. The method can be used in situations where returning all valid scopes from the get_original_scopes is not practical.

Parameters

- **request_scopes** – A list of scopes that were requested by client
- **refresh_token** – Unicode refresh_token
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Refresh token grant

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

Parameters

- **token** – The token string.
- **token_type_hint** – access_token or refresh_token.
- **request** – The HTTP Request (oauthlib.common.Request)

Method is used by:

- Revocation Endpoint

rotate_refresh_token(*request*)
Determine whether to rotate the refresh token. Default, yes.

When access tokens are refreshed the old refresh token can be kept or replaced with a new one (rotated). Return True to rotate and and False for keeping original.

Parameters **request** – oauthlib.common.Request

Return type True or False

Method is used by:

- Refresh Token Grant

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

The code should at minimum be stored with:

- the client_id (client_id)
- the redirect URI used (request.redirect_uri)
- a resource owner / user (request.user)
- the authorized scopes (request.scopes)
- the client state, if given (code.get('state'))

The 'code' argument is actually a dictionary, containing at least a 'code' key with the actual authorization code:

{'code': 'sdf345jsdf0934f'}

It may also have a 'state' key containing a nonce for the client, if it chose to send one. That value should be saved and used in 'validate_code'.

It may also have a 'claims' parameter which, when present, will be a dict deserialized from JSON as described at http://openid.net/specs/openid-connect-core-1_0.html#ClaimsParameter This value should be saved in this method and used again in 'validate_code'.

Parameters

- **client_id** – Unicode client identifier
- **code** – A dict of the authorization code grant and, optionally, state.
- **request** – The HTTP Request (oauthlib.common.Request)

Method is used by:

- Authorization Code Grant

save_bearer_token(*token*, *request*, **args*, ***kwargs*)

Persist the Bearer token.

The Bearer token should at minimum be associated with:

- a client and it's client_id, if available
- a resource owner / user (request.user)
- authorized scopes (request.scopes)
- an expiration time
- a refresh token, if issued
- a claims document, if present in request.claims

The Bearer token dict may hold a number of items:

```
{
    'token_type': 'Bearer',
    'access_token': 'askfjh234as9sd8',
    'expires_in': 3600,
    'scope': 'string of space separated authorized scopes',
    'refresh_token': '23sdf876234',  # if issued
    'state': 'given_by_client',  # if supplied by client
}
```

Note that while "scope" is a string-separated list of authorized scopes, the original list is still available in request.scopes

Also note that if an Authorization Code grant request included a valid claims parameter (for OpenID Connect) then the request.claims property will contain the claims dict, which should be saved for later use when generating the id_token and/or UserInfo response content.

Parameters

- **client_id** – Unicode client identifier
- **token** – A Bearer token dict

- **request** – The HTTP Request (oauthlib.common.Request)

Return type The default redirect URI for the client

Method is used by all core grant types issuing Bearer tokens:

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials Grant (might not associate a client)
- Client Credentials grant

save_token(*token*, *request*, **args*, ***kwargs*)
Persist the token with a token type specific method.

Currently, only save_bearer_token is supported.

validate_bearer_token(*token*, *scopes*, *request*)
Ensure the Bearer token is valid and authorized access to scopes.

Parameters

- **token** – A string of random characters.
- **scopes** – A list of scopes associated with the protected resource.
- **request** – The HTTP Request (oauthlib.common.Request)

A key to OAuth 2 security and restricting impact of leaked tokens is the short expiration time of tokens, *always ensure the token has not expired!*.

Two different approaches to scope validation:

1. **all(scopes). The token must be authorized access to all scopes** associated with the resource. For example, the token has access to `read-only` and `images`, thus the client can view images but not upload new. Allows for fine grained access control through combining various scopes.
2. **any(scopes). The token must be authorized access to one of the** scopes associated with the resource. For example, token has access to `read-only-images`. Allows for fine grained, although arguably less convenient, access control.

A powerful way to use scopes would mimic UNIX ACLs and see a scope as a group with certain privileges. For a restful API these might map to HTTP verbs instead of read, write and execute.

Note, the request.user attribute can be set to the resource owner associated with this token. Similarly the request.client and request.scopes attribute can be set to associated client object and authorized scopes. If you then use a decorator such as the one provided for django these attributes will be made available in all protected views as keyword arguments.

Parameters

- **token** – Unicode Bearer token
- **scopes** – List of scopes (defined by you)
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is indirectly used by all core Bearer token issuing grant types:

- Authorization Code Grant
- Implicit Grant

- Resource Owner Password Credentials Grant
- Client Credentials Grant

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

Note, while not strictly necessary it can often be very convenient to set request.client to the client object associated with the given client_id.

Parameters **request** – oauthlib.common.Request

Return type True or False

Method is used by:

- Authorization Code Grant
- Implicit Grant

validate_code(*client_id*, *code*, *client*, *request*, **args*, ***kwargs*)
Verify that the authorization_code is valid and assigned to the given client.

Before returning true, set the following based on the information stored with the code in 'save_authorization_code':

•request.user

•request.state (if given)

•request.scopes

•request.claims (if given)

OBS! The request.user attribute should be set to the resource owner associated with this authorization code. Similarly request.scopes must also be set.

The request.claims property, if it was given, should assigned a dict.

Parameters

- **client_id** – Unicode client identifier
- **code** – Unicode authorization code
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant

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

Parameters

- **client_id** – Unicode client identifier
- **grant_type** – Unicode grant type, i.e. authorization_code, password.
- **client** – Client object set by you, see authenticate_client.

- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant
- Resource Owner Password Credentials Grant
- Client Credentials Grant
- Refresh Token Grant

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

All clients should register the absolute URIs of all URIs they intend to redirect to. The registration is outside of the scope of oauthlib.

Parameters

- **client_id** – Unicode client identifier
- **redirect_uri** – Unicode absolute URI
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant
- Implicit Grant

validate_refresh_token(*refresh_token*, *client*, *request*, **args*, ***kwargs*)
Ensure the Bearer token is valid and authorized access to scopes.

OBS! The request.user attribute should be set to the resource owner associated with this refresh token.

Parameters

- **refresh_token** – Unicode refresh token
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant (indirectly by issuing refresh tokens)
- Resource Owner Password Credentials Grant (also indirectly)
- Refresh Token Grant

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

Parameters

- **client_id** – Unicode client identifier

- **response_type** – Unicode response type, i.e. code, token.
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Authorization Code Grant
- Implicit Grant

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

Parameters

- **client_id** – Unicode client identifier
- **scopes** – List of scopes (defined by you)
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by all core grant types:

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials Grant
- Client Credentials Grant

validate_silent_authorization(*request*)
Ensure the logged in user has authorized silent OpenID authorization.

Silent OpenID authorization allows access tokens and id tokens to be granted to clients without any user prompt or interaction.

Parameters **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

validate_silent_login(*request*)
Ensure session user has authorized silent OpenID login.

If no user is logged in or has not authorized silent login, this method should return False.

If the user is logged in but associated with multiple accounts and not selected which one to link to the token then this method should raise an oauthlib.oauth2.AccountSelectionRequired error.

Parameters **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

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

OBS! The validation should also set the user attribute of the request to a valid resource owner, i.e. request.user = username or similar. If not set you will be unable to associate a token with a user in the persistance method used (commonly, save_bearer_token).

Parameters

- **username** – Unicode username
- **password** – Unicode password
- **client** – Client object set by you, see authenticate_client.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- Resource Owner Password Credentials Grant

validate_user_match(*id_token_hint*, *scopes*, *claims*, *request*)
Ensure client supplied user id hint matches session user.

If the sub claim or id_token_hint is supplied then the session user must match the given ID.

Parameters

- **id_token_hint** – User identifier string.
- **scopes** – List of OAuth 2 scopes and OpenID claims (strings).
- **claims** – OpenID Connect claims dict.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

3. Create your composite endpoint

Each of the endpoints can function independently from each other, however for this example it is easier to consider them as one unit. An example of a pre-configured all-in-one Authorization Code Grant endpoint is given below.

```
# From the previous section on validators
from my_validator import MyRequestValidator

from oauthlib.oauth2 import WebApplicationServer

validator = MyRequestValidator()
server = WebApplicationServer(validator)
```

Relevant sections include:

Preconfigured all-in-one servers

A pre configured server is an all-in-one endpoint serving a specific class of application clients. As the individual endpoints, they depend on the use of a *Request Validator*.

Construction is simple, only import your validator and you are good to go:

```
from your_validator import your_validator
from oauthlib.oauth2 import WebApplicationServer

server = WebApplicationServer(your_validator)
```

If you prefer to construct tokens yourself you may pass a token generator:

```
def your_token_generator(request, refresh_token=False):
    return 'a_custom_token' + request.client_id

server = WebApplicationServer(your_validator, token_generator=your_token_generator)
```

This function is passed the request object and a boolean indicating whether to generate an access token (False) or a refresh token (True).

class oauthlib.oauth2.**WebApplicationServer**(*request_validator*, *token_generator=None*, *token_expires_in=None*, *refresh_token_generator=None*, ***kwargs*)

An all-in-one endpoint featuring Authorization code grant and Bearer tokens.

class oauthlib.oauth2.**MobileApplicationServer**(*request_validator*, *token_generator=None*, *token_expires_in=None*, *refresh_token_generator=None*, ***kwargs*)

An all-in-one endpoint featuring Implicit code grant and Bearer tokens.

class oauthlib.oauth2.**LegacyApplicationServer**(*request_validator*, *token_generator=None*, *token_expires_in=None*, *refresh_token_generator=None*, ***kwargs*)

An all-in-one endpoint featuring Resource Owner Password Credentials grant and Bearer tokens.

class oauthlib.oauth2.**BackendApplicationServer**(*request_validator*, *token_generator=None*, *token_expires_in=None*, *refresh_token_generator=None*, ***kwargs*)

An all-in-one endpoint featuring Client Credentials grant and Bearer tokens.

4. Create your endpoint views

We are implementing support for the Authorization Code Grant and will therefore need two views for the authorization, pre- and post-authorization together with the token view. We also include an error page to redirect users to if the client supplied invalid credentials in their redirection, for example an invalid redirect URI.

The example using Django but should be transferable to any framework.

```
# Handles GET and POST requests to /authorize
class AuthorizationView(View):

    def __init__(self):
        # Using the server from previous section
        self._authorization_endpoint = server

    def get(self, request):
        # You need to define extract_params and make sure it does not
        # include file like objects waiting for input. In Django this
        # is request.META['wsgi.input'] and request.META['wsgi.errors']
        uri, http_method, body, headers = extract_params(request)

        try:
            scopes, credentials = self._authorization_endpoint.validate_authorization_
↪request(
                uri, http_method, body, headers)

            # Not necessarily in session but they need to be
            # accessible in the POST view after form submit.
            request.session['oauth2_credentials'] = credentials

            # You probably want to render a template instead.
            response = HttpResponse()
            response.write('<h1> Authorize access to %s </h1>' % client_id)
            response.write('<form method="POST" action="/authorize">')
            for scope in scopes or []:
                response.write('<input type="checkbox" name="scopes" ' +
                'value="%s"/> %s' % (scope, scope))
                response.write('<input type="submit" value="Authorize"/>')
            return response

        # Errors that should be shown to the user on the provider website
        except errors.FatalClientError as e:
            return response_from_error(e)

        # Errors embedded in the redirect URI back to the client
        except errors.OAuth2Error as e:
            return HttpResponseRedirect(e.in_uri(e.redirect_uri))

    @csrf_exempt
    def post(self, request):
        uri, http_method, body, headers = extract_params(request)

        # The scopes the user actually authorized, i.e. checkboxes
        # that were selected.
        scopes = request.POST.getlist(['scopes'])

        # Extra credentials we need in the validator
        credentials = {'user': request.user}

        # The previously stored (in authorization GET view) credentials
        credentials.update(request.session.get('oauth2_credentials', {}))

        try:
            headers, body, status = self._authorization_endpoint.create_authorization_
↪response(
```

```
            uri, http_method, body, headers, scopes, credentials)
            return response_from_return(headers, body, status)

        except errors.FatalClientError as e:
            return response_from_error(e)

# Handles requests to /token
class TokenView(View):

    def __init__(self):
        # Using the server from previous section
        self._token_endpoint = server

    def post(self, request):
        uri, http_method, body, headers = extract_params(request)

        # If you wish to include request specific extra credentials for
        # use in the validator, do so here.
        credentials = {'foo': 'bar'}

        headers, body, status = self._token_endpoint.create_token_response(
                uri, http_method, body, headers, credentials)

        # All requests to /token will return a json response, no redirection.
        return response_from_return(headers, body, status)

def response_from_return(headers, body, status):
    response = HttpResponse(content=body, status=status)
    for k, v in headers.items():
        response[k] = v
    return response

def response_from_error(e)
    return HttpResponseBadRequest('Evil client is unable to send a proper request.␣
↪Error is: ' + e.description)
```

5. Protect your APIs using scopes

Let's define a decorator we can use to protect the views.

```
class OAuth2ProviderDecorator(object):

    def __init__(self, resource_endpoint):
        self._resource_endpoint = resource_endpoint

    def protected_resource_view(self, scopes=None):
        def decorator(f):
            @functools.wraps(f)
            def wrapper(request):
                # Get the list of scopes
                try:
                    scopes_list = scopes(request)
                except TypeError:
                    scopes_list = scopes

                uri, http_method, body, headers = extract_params(request)
```

```
                valid, r = self._resource_endpoint.verify_request(
                        uri, http_method, body, headers, scopes_list)

                # For convenient parameter access in the view
                add_params(request, {
                    'client': r.client,
                    'user': r.user,
                    'scopes': r.scopes
                })
                if valid:
                    return f(request)
                else:
                    # Framework specific HTTP 403
                    return HttpResponseForbidden()
            return wrapper
        return decorator

provider = OAuth2ProviderDecorator(server)
```

At this point you are ready to protect your API views with OAuth. Take some time to come up with a good set of scopes as they can be very powerful in controlling access.

```
@provider.protected_resource_view(scopes=['images'])
def i_am_protected(request, client, resource_owner):
    # One of your many OAuth 2 protected resource views
    # Returns whatever you fancy
    # May be bound to various scopes of your choosing
    return HttpResponse('pictures of cats')
```

The set of scopes that protects a view may also be dynamically configured at runtime by a function, rather then by a list.

```
def dynamic_scopes(request):
    # Place code here to dynamically determine the scopes
    # and return as a list
    return ['images']

@provider.protected_resource_view(scopes=dynamic_scopes)
def i_am_also_protected(request, client, resource_owner, **kwargs)
    # A view that has its views functionally set.
    return HttpResponse('pictures of cats')
```

6. Let us know how it went!

Drop a line in our G+ community or open a GitHub issue =)

If you run into issues it can be helpful to enable debug logging.

```
import logging
import sys
log = logging.getLogger('oauthlib')
log.addHandler(logging.StreamHandler(sys.stdout))
log.setLevel(logging.DEBUG)
```

Provider Endpoints

Endpoints in OAuth 2 are targets with a specific responsibility and often associated with a particular URL. Because of this the word endpoint might be used interchangably from the endpoint url.

The main three responsibilities in an OAuth 2 flow is to authorize access to a certain users resources to a client, to supply said client with a token embodying this authorization and to verify that the token is valid when the client attempts to access the user resources on their behalf.

Authorization

Authorization can be either explicit or implicit. The former require the user to actively authorize the client by being redirected to the authorization endpoint. There he/she is usually presented by a form and asked to either accept or deny access to certain scopes. These scopes can be thought of as Access Control Lists that are tied to certain privileges and categories of resources, such as write access to their status feed or read access to their profile. It is vital that the implications of granting access to a certain scope is very clear in the authorization form presented to the user. It is up to the provider to allow the user agree to all, a few or none of the scopes. Being flexible here is a great benefit to the user at the cost of added complexity in both the provider and clients.

Implicit authorization happens when the authorization happens before the OAuth flow, such as the user giving the client his/her password and username, or if there is a very high level of trust between the user, client and provider and no explicit authorization is necessary.

Examples of explicit authorization is the Authorization Code Grant and the Implicit Grant.

Examples of implicit authorization is the Resource Owner Password Credentials Grant and the Client Credentials Grant.

Pre Authorization Request OAuth is known for it's authorization page where the user accepts or denies access to a certain client and set of scopes. Before presenting the user with such a form you need to ensure the credentials the client supplied in the redirection to this page are valid.

```
# Initial setup
from your_validator import your_validator
server = WebApplicationServer(your_validator)

# Validate request
uri = 'https://example.com/authorize?client_id=foo&state=xyz
headers, body, http_method = {}, '', 'GET'

from oauthlib.oauth2 import FatalClientError
from your_framework import redirect
try:
    scopes, credentials = server.validate_authorization_request(
        uri, http_method, body, headers)
    # scopes will hold default scopes for client, i.e.
    ['https://example.com/userProfile', 'https://example.com/pictures']

    # credentials is a dictionary of
    {
        'client_id': 'foo',
        'redirect_uri': 'https://foo.com/welcome_back',
        'response_type': 'code',
        'state': 'randomstring',
    }
    # these credentials will be needed in the post authorization view and
    # should be persisted between. None of them are secret but take care
```

```
    # to ensure their integrity if embedding them in the form or cookies.
    from your_datastore import persist_credentials
    persist_credentials(credentials)

    # Present user with a nice form where client (id foo) request access to
    # his default scopes (omitted from request), after which you will
    # redirect to his default redirect uri (omitted from request).

except FatalClientError as e:
    # this is your custom error page
    from your_view_helpers import error_to_response
    return error_to_response(e)
```

Post Authorization Request Generally, this is where you handle the submitted form. Rather than using `validate_authorization_request` we use `create_authorization_response` which in the case of Authorization Code Grant embed an authorization code in the client provided redirect uri.

```
# Initial setup
from your_validator import your_validator
server = WebApplicationServer(your_validator)

# Validate request
uri = 'https://example.com/post_authorize?client_id=foo
headers, body, http_method = {}, '', 'GET'

# Fetch the credentials saved in the pre authorization phase
from your_datastore import fetch_credentials
credentials = fetch_credentials()

# Fetch authorized scopes from the request
from your_framework import request
scopes = request.POST.get('scopes')

from oauthlib.oauth2 import FatalClientError, OAuth2Error
from your_framework import http_response
http_response(body, status=status, headers=headers)
try:
    headers, body, status = server.create_authorization_response(
        uri, http_method, body, headers, scopes, credentials)
    # headers = {'Location': 'https://foo.com/welcome_back?code=somerandomstring&
↪state=xyz'}, this might change to include suggested headers related
    # to cache best practices etc.
    # body = '', this might be set in future custom grant types
    # status = 302, suggested HTTP status code

    return http_response(body, status=status, headers=headers)

except FatalClientError as e:
    # this is your custom error page
    from your_view_helpers import error_to_response
    return error_to_response(e)

except OAuth2Error as e:
    # Less grave errors will be reported back to client
    client_redirect_uri = credentials.get('redirect_uri')
    redirect(e.in_uri(client_redirect_uri))
```

class oauthlib.oauth2.**AuthorizationEndpoint**(*default_response_type*, *default_token_type*, *response_types*)

Authorization endpoint - used by the client to obtain authorization from the resource owner via user-agent redirection.

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server MUST first verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g. username and password login, session cookies) is beyond the scope of this specification.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component, which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component:

```
https://example.com/path?query=component             # OK
https://example.com/path?query=component#fragment    # Not OK
```

Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the authorization endpoint:

```
# We will deny any request which URI schema is not with https
```

The authorization server MUST support the use of the HTTP "GET" method [RFC2616] for the authorization endpoint, and MAY support the use of the "POST" method as well:

```
# HTTP method is currently not enforced
```

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once:

```
# Enforced through the design of oauthlib.common.Request
```

create_authorization_response(*endpoint*, *uri*, **args*, ***kwargs*)

Extract response_type and route to the designated handler.

validate_authorization_request(*endpoint*, *uri*, **args*, ***kwargs*)

Extract response_type and route to the designated handler.

Token creation

Token endpoints issue tokens to clients who have already been authorized access, be it by explicit actions from the user or implicitly. The token response is well defined and typically consists of an unguessable access token, the token type, its expiration from now in seconds, and depending on the scenario, a refresh token to be used to fetch new access tokens without authorization.

One argument for OAuth 2 being more scalable than OAuth 1 is that tokens may contain hidden information. A provider may embed information such as client identifier, user identifier, expiration times, etc. in the token by encrypting it. This trades a slight increase in work required to decrypt the token but frees the necessary database lookups otherwise required, thus improving latency substantially. OAuthlib currently does not provide a method for creating crypto-tokens but may do in the future.

The standard token type, Bearer, does not require that the provider bind a specific client to the token. Not binding clients to tokens allow for anonymized tokens which unless you are certain you need them, are a bad idea.

Token Request A POST request used in most grant types but with a varied setup of credentials. If you wish to embed extra credentials in the request, i.e. for later use in validation or when creating the token, you can use the `credentials` argument in `create_token_response`.

All responses are in json format and the headers argument returned by `create_token_response` will contain a few suggested headers related to content type and caching.

```
# Initial setup
from your_validator import your_validator
server = WebApplicationServer(your_validator)

# Validate request
uri = 'https://example.com/token'
http_method = 'POST'
body = 'code=somerandomstring&'
       'grant_type=authorization_code&'
# Clients authenticate through a method of your choosing, for example
# using HTTP Basic Authentication
headers = { 'Authorization': 'Basic ksjdhf923sf' }

# Extra credentials you wish to include
credentials = {'client_ip': '1.2.3.4'}

headers, body, status = server.create_token_response(
    uri, http_method, body, headers, credentials)

# headers will contain some suggested headers to add to your response
{
    'Content-Type': 'application/json',
    'Cache-Control': 'no-store',
    'Pragma': 'no-cache',
}
# body will contain the token in json format and expiration from now
# in seconds.
{
    'access_token': 'sldafh309sdf',
    'refresh_token': 'alsounguessablerandomstring',
    'expires_in': 3600,
    'scope': 'https://example.com/userProfile https://example.com/pictures',
    'token_type': 'Bearer'
}
# body will contain an error code and possibly an error description if
# the request failed, also in json format.
{
    'error': 'invalid_grant_type',
    'description': 'athorizatoin_coed is not a valid grant type'
}
# status will be a suggested status code, 200 on ok, 400 on bad request
# and 401 if client is trying to use an invalid authorization code,
# fail to authenticate etc.

from your_framework import http_response
http_response(body, status=status, headers=headers)
```

class `oauthlib.oauth2.`**`TokenEndpoint`**(*default_grant_type*, *default_token_type*, *grant_types*)
Token issuing endpoint.

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an

access token is issued directly).

The means through which the client obtains the location of the token endpoint are beyond the scope of this specification, but the location is typically provided in the service documentation.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component, which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component:

```
https://example.com/path?query=component             # OK
https://example.com/path?query=component#fragment    # Not OK
```

Since requests to the authorization endpoint result in user Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the token endpoint:

```
# We will deny any request which URI schema is not with https
```

The client MUST use the HTTP "POST" method when making access token requests:

```
# HTTP method is currently not enforced
```

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once:

```
# Delegated to each grant type.
```

create_token_response(*endpoint*, *uri*, **args*, ***kwargs*)
Extract grant_type and route to the designated handler.

Resource authorization

Resource endpoints verify that the token presented is valid and granted access to the scopes associated with the resource in question.

Request Verfication Each view may set certain scopes under which it is bound. Only requests that present an access token bound to the correct scopes may access the view. Access tokens are commonly embedded in the authorization header but may appear in the query or the body as well.

```
# Initial setup
from your_validator import your_validator
server = WebApplicationServer(your_validator)

# Per view scopes
required_scopes = ['https://example.com/userProfile']

# Validate request
uri = 'https://example.com/userProfile?access_token=sldafh309sdf'
headers, body, http_method = {}, '', 'GET'

valid, oauthlib_request = server.verify_request(
    uri, http_method, body, headers, required_scopes)

# oauthlib_request has a few convenient attributes set such as
# oauthlib_request.client = the client associated with the token
# oauthlib_request.user = the user associated with the token
```

```
# oauthlib_request.scopes = the scopes bound to this token

if valid:
    # return the protected resource / view
else:
    # return an http forbidden 403
```

class oauthlib.oauth2.**ResourceEndpoint**(*default_token*, *token_types*)

Authorizes access to protected resources.

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the access token and ensure that it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification but generally involve an interaction or coordination between the resource server and the authorization server:

```
# For most cases, returning a 403 should suffice.
```

The method in which the client utilizes the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP "Authorization" request header field [RFC2617] with an authentication scheme defined by the specification of the access token type used, such as [RFC6750]:

```
# Access tokens may also be provided in query and body
https://example.com/protected?access_token=kjfch2345sdf   # Query
access_token=sdf23409df   # Body
```

find_token_type(*request*)

Token type identification.

RFC 6749 does not provide a method for easily differentiating between different token types during protected resource access. We estimate the most likely token type (if any) by asking each known token type to give an estimation based on the request.

verify_request(*endpoint*, *uri*, **args*, ***kwargs*)

Validate client, code etc, return body + headers

Token revocation

Revocation endpoints invalidate access and refresh tokens upon client request. They are commonly part of the authorization endpoint.

```
# Initial setup
from your_validator import your_validator
server = WebApplicationServer(your_validator)

# Token revocation
uri = 'https://example.com/revoke_token'
headers, body, http_method = {}, 'token=sldafh309sdf', 'POST'

headers, body, status = server.create_revocation_response(uri,
    headers=headers, body=body, http_method=http_method)

from your_framework import http_response
http_response(body, status=status, headers=headers)
```

class oauthlib.oauth2.**RevocationEndpoint**(*request_validator*, *supported_token_types=None*, *enable_jsonp=False*)

Token revocation endpoint.

Endpoint used by authenticated clients to revoke access and refresh tokens. Commonly this will be part of the Authorization Endpoint.

create_revocation_response(*endpoint*, *uri*, **args*, ***kwargs*)

Revoke supplied access or refresh token.

The authorization server responds with HTTP status code 200 if the token has been revoked sucessfully or if the client submitted an invalid token.

Note: invalid tokens do not cause an error response since the client cannot handle such an error in a reasonable way. Moreover, the purpose of the revocation request, invalidating the particular token, is already achieved.

The content of the response body is ignored by the client as all necessary information is conveyed in the response code.

An invalid token type hint value is ignored by the authorization server and does not influence the revocation response.

validate_revocation_request(*request*)

Ensure the request is valid.

The client constructs the request by including the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

token (REQUIRED). The token that the client wants to get revoked.

token_type_hint (OPTIONAL). A hint about the type of the token submitted for revocation. Clients MAY pass this parameter in order to help the authorization server to optimize the token lookup. If the server is unable to locate the token using the given hint, it MUST extend its search accross all of its supported token types. An authorization server MAY ignore this parameter, particularly if it is able to detect the token type automatically. This specification defines two such values:

- **access_token: An Access Token as defined in [RFC6749],** section 1.4
- **refresh_token: A Refresh Token as defined in [RFC6749],** section 1.5

Specific implementations, profiles, and extensions of this specification MAY define other values for this parameter using the registry defined in Section 4.1.2.

The client also includes its authentication credentials as described in Section 2.3. of [RFC6749].

There are three different endpoints, the authorization endpoint which mainly handles user authorization, the token endpoint which provides tokens and the resource endpoint which provides access to protected resources. It is to the endpoints you will feed requests and get back an almost complete response. This process is simplified for you using a decorator such as the django one described later.

The main purpose of the endpoint in OAuthLib is to figure out which grant type or token to dispatch the request to.

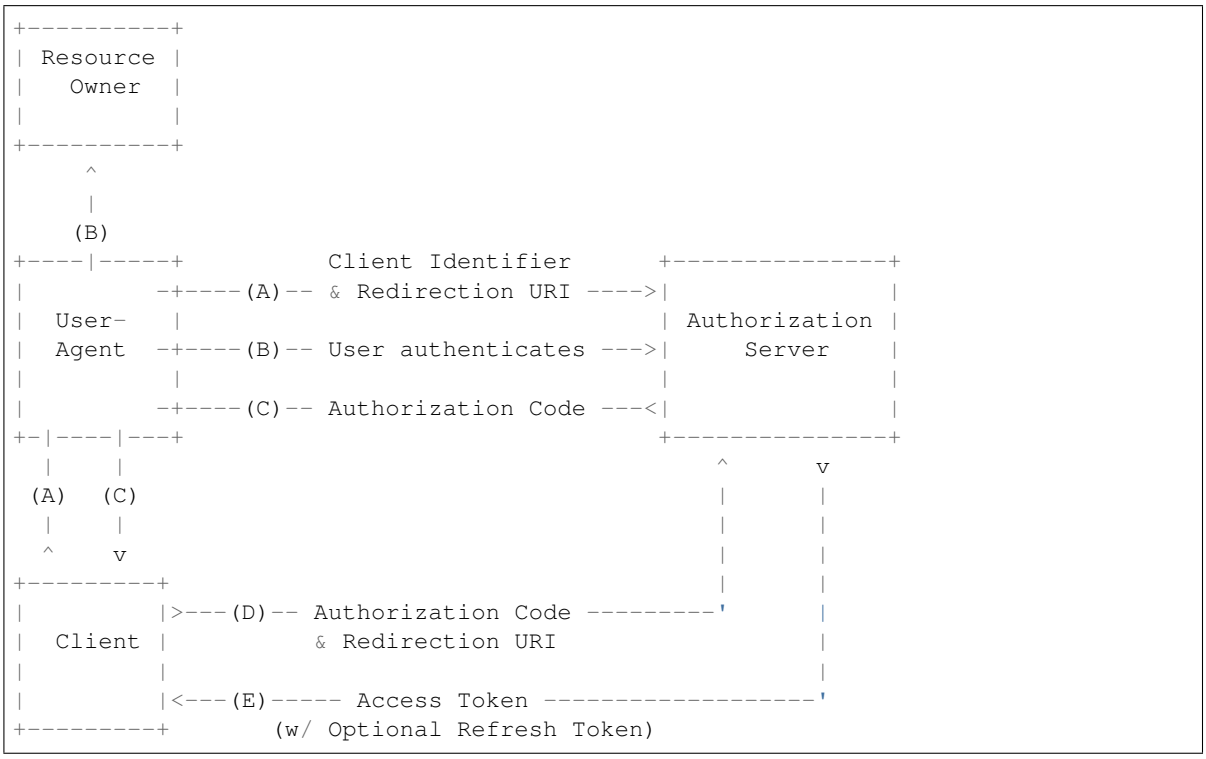
Grant types

Authorization Code Grant

class oauthlib.oauth2.**AuthorizationCodeGrant**(*request_validator=None*, ***kwargs*)

Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server:



Note: The lines illustrating steps (A), (B), and (C) are broken into two parts as they pass through the user-agent.

Figure 3: Authorization Code Flow

The flow illustrated in Figure 3 includes the following steps:

1. The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).
2. The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.
3. Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.
4. The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.
5. The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

create_authorization_code(*request*)
Generates an authorization grant represented as a dictionary.

create_authorization_response(*request*, *token_handler*)

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format, per Appendix B:

response_type REQUIRED. Value MUST be set to "code" for standard OAuth2 authorization flow. For OpenID Connect it must be one of "code token", "code id_token", or "code token id_token" - we essentially test that "code" appears in the response_type.

client_id REQUIRED. The client identifier as described in Section 2.2.

redirect_uri OPTIONAL. As described in Section 3.1.2.

scope OPTIONAL. The scope of the access request as described by Section 3.3.

state RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

Parameters

- **request** – oauthlib.commong.Request
- **token_handler** – A token handler instace, for example of type oauthlib.oauth2.BearerToken.

Returns headers, body, status

Raises FatalClientError on invalid redirect URI or client id. ValueError if scopes are not set on the request object.

A few examples:

```
>>> from your_validator import your_validator
>>> request = Request('https://example.com/authorize?client_id=valid'
...                   '&redirect_uri=http%3A%2F%2Fclient.com%2F')
>>> from oauthlib.common import Request
>>> from oauthlib.oauth2 import AuthorizationCodeGrant, BearerToken
>>> token = BearerToken(your_validator)
>>> grant = AuthorizationCodeGrant(your_validator)
>>> grant.create_authorization_response(request, token)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "oauthlib/oauth2/rfc6749/grant_types.py", line 513, in create_
↪authorization_response
        raise ValueError('Scopes must be set on post auth.')
ValueError: Scopes must be set on post auth.
>>> request.scopes = ['authorized', 'in', 'some', 'form']
>>> grant.create_authorization_response(request, token)
(u'http://client.com/?error=invalid_request&error_
↪description=Missing+response_type+parameter.', None, None, 400)
>>> request = Request('https://example.com/authorize?client_id=valid'
...                   '&redirect_uri=http%3A%2F%2Fclient.com%2F'
...                   '&response_type=code')
>>> request.scopes = ['authorized', 'in', 'some', 'form']
>>> grant.create_authorization_response(request, token)
(u'http://client.com/?code=u3F05aEObJuP2k7DordviIgW5wl52N', None, None, 200)
>>> # If the client id or redirect uri fails validation
>>> grant.create_authorization_response(request, token)
```

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "oauthlib/oauth2/rfc6749/grant_types.py", line 515, in create_
↪authorization_response
        >>> grant.create_authorization_response(request, token)
    File "oauthlib/oauth2/rfc6749/grant_types.py", line 591, in validate_
↪authorization_request
oauthlib.oauth2.rfc6749.errors.InvalidClientIdError
```

create_token_response(*request*, *token_handler*)
Validate the authorization code.

The client MUST NOT use the authorization code more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

prepare_authorization_response(*request*, *token*, *headers*, *body*, *status*)
Place token according to response mode.

Base classes can define a default response mode for their authorization response by overriding the static *default_response_mode* member.

validate_authorization_request(*request*)
Check the authorization request for normal and fatal errors.

A normal error could be a missing response_type parameter or the client attempting to access scope it is not allowed to ask authorization for. Normal errors can safely be included in the redirection URI and sent back to the client.

Fatal errors occur when the client_id or redirect_uri is invalid or missing. These must be caught by the provider and handled, how this is done is outside of the scope of OAuthLib but showing an error page describing the issue is a good idea.

Implicit Grant

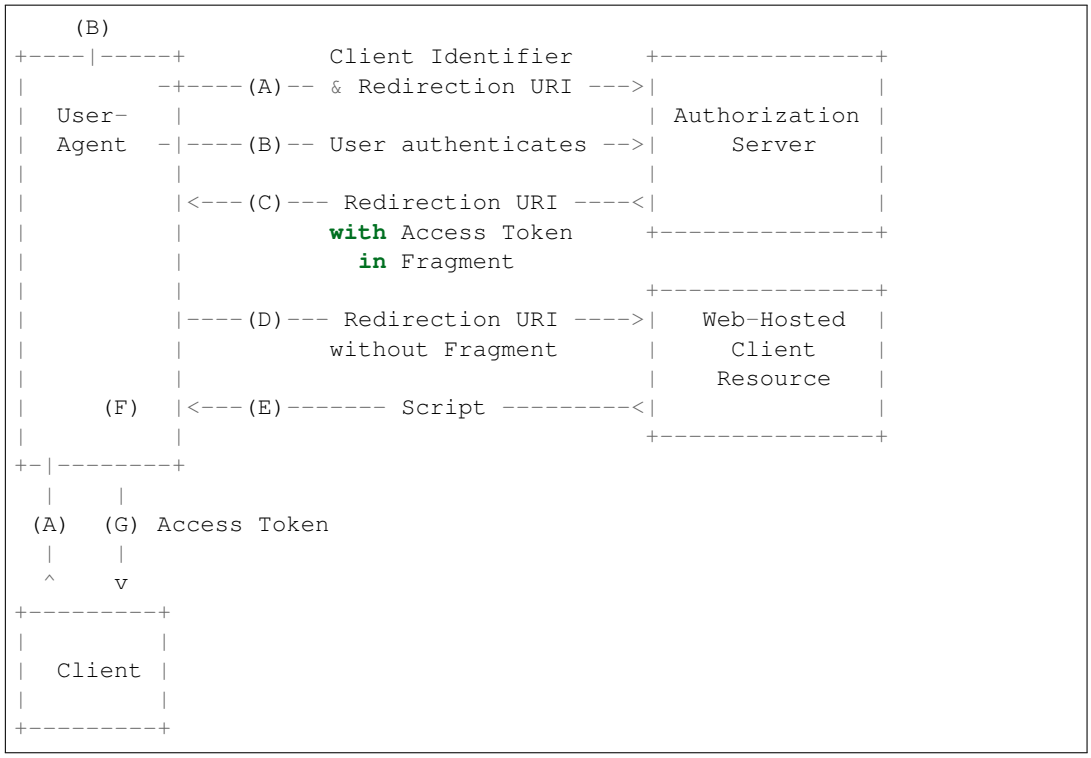
class oauthlib.oauth2.**ImplicitGrant**(*request_validator=None*, ***kwargs*)
Implicit Grant

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

Unlike the authorization code grant type, in which the client makes separate requests for authorization and for an access token, the client receives the access token as the result of the authorization request.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device:

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     ^
     |
```



Note: The lines illustrating steps (A) and (B) are broken into two parts as they pass through the user-agent.

Figure 4: Implicit Grant Flow

The flow illustrated in Figure 4 includes the following steps:

1. The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).
2. The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.
3. Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.
4. The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment per [RFC2616]). The user-agent retains the fragment information locally.
5. The web-hosted client resource returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.
6. The user-agent executes the script provided by the web-hosted client resource locally, which extracts the access token.
7. The user-agent passes the access token to the client.

   See Section 10.3 and Section 10.16 for important security considerations when using the implicit grant.

create_authorization_response(*request*, *token_handler*)

Create an authorization response. The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format, per Appendix B:

response_type

REQUIRED. Value MUST be set to "token" for standard OAuth2 implicit flow or "id_token token" or just "id_token" for OIDC implicit flow

client_id REQUIRED. The client identifier as described in Section 2.2.

redirect_uri OPTIONAL. As described in Section 3.1.2.

scope OPTIONAL. The scope of the access request as described by Section 3.3.

state RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

The authorization server validates the request to ensure that all required parameters are present and valid. The authorization server MUST verify that the redirection URI to which it will redirect the access token matches a redirection URI registered by the client as described in Section 3.1.2.

create_token_response(*request*, *token_handler*)

Return token or error embedded in the URI fragment.

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

access_token REQUIRED. The access token issued by the authorization server.

token_type REQUIRED. The type of the token issued as described in Section 7.1. Value is case insensitive.

expires_in RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

scope OPTIONAL, if identical to the scope requested by the client; otherwise, REQUIRED. The scope of the access token as described by Section 3.3.

state REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

The authorization server MUST NOT issue a refresh token.

prepare_authorization_response(*request*, *token*, *headers*, *body*, *status*)

Place token according to response mode.

Base classes can define a default response mode for their authorization response by overriding the static *default_response_mode* member.

validate_token_request(*request*)

Check the token request for normal and fatal errors.

This method is very similar to validate_authorization_request in the AuthorizationCodeGrant but differ in a few subtle areas.

A normal error could be a missing response_type parameter or the client attempting to access scope it is not allowed to ask authorization for. Normal errors can safely be included in the redirection URI and sent back to the client.

Fatal errors occur when the client_id or redirect_uri is invalid or missing. These must be caught by the provider and handled, how this is done is outside of the scope of OAuthLib but showing an error page describing the issue is a good idea.

Resource Owner Password Credentials Grant

class oauthlib.oauth2.**ResourceOwnerPasswordCredentialsGrant**(*request_validator=None*, ***kwargs*)

Resource Owner Password Credentials Grant

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. The authorization server should take special care when enabling this grant type and only allow it when other flows are not viable.

This grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token:

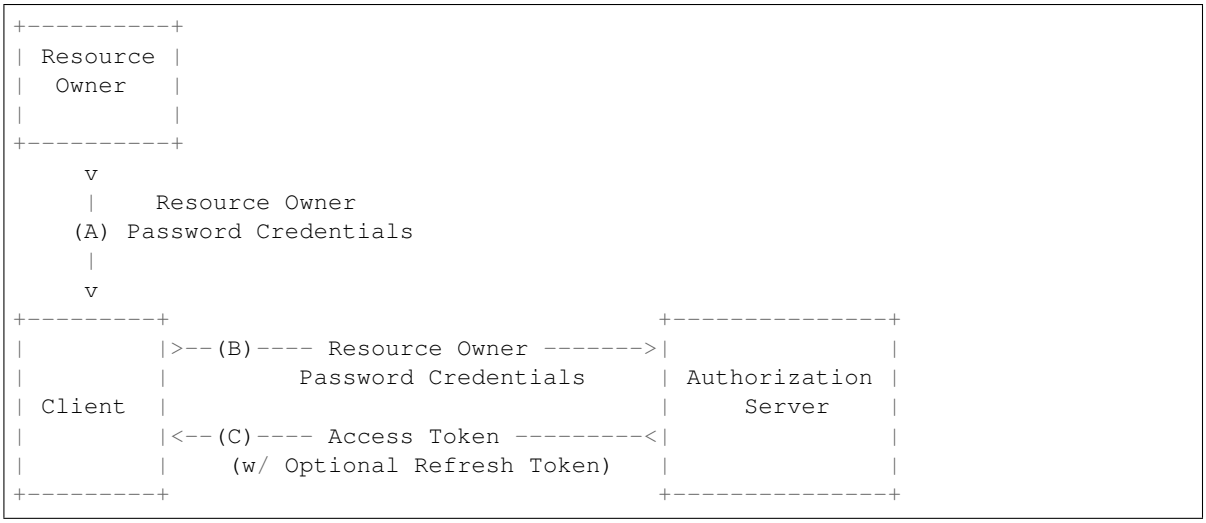


Figure 5: Resource Owner Password Credentials Flow

The flow illustrated in Figure 5 includes the following steps:

1. **The resource owner provides the client with its username and** password.
2. **The client requests an access token from the authorization** server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.
3. **The authorization server authenticates the client and validates** the resource owner credentials, and if valid, issues an access token.

create_token_response(*request*, *token_handler*)

Return token or error in json format.

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

prepare_authorization_response(*request*, *token*, *headers*, *body*, *status*)
Place token according to response mode.

Base classes can define a default response mode for their authorization response by overriding the static *default_response_mode* member.

validate_token_request(*request*)
The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type REQUIRED. Value MUST be set to "password".

username REQUIRED. The resource owner username.

password REQUIRED. The resource owner password.

scope OPTIONAL. The scope of the access request as described by Section 3.3.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

The authorization server MUST:

o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),

o authenticate the client if client authentication is included, and

o validate the resource owner password credentials using its existing password validation algorithm.

Since this access token request utilizes the resource owner's password, the authorization server MUST protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts).

Client Credentials Grant

class oauthlib.oauth2.**ClientCredentialsGrant**(*request_validator=None*, ***kwargs*)
Client Credentials Grant

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients:

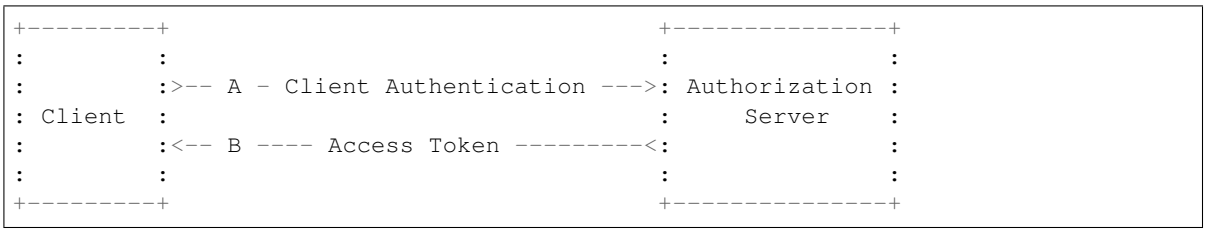


Figure 6: Client Credentials Flow

The flow illustrated in Figure 6 includes the following steps:

1. **The client authenticates with the authorization server and** requests an access token from the token endpoint.
2. **The authorization server authenticates the client, and if valid,** issues an access token.

create_token_response(*request*, *token_handler*)
Return token or error in JSON format.

If the access token request is valid and authorized, the authorization server issues an access token as described in Section 5.1. A refresh token SHOULD NOT be included. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

prepare_authorization_response(*request*, *token*, *headers*, *body*, *status*)
Place token according to response mode.

Base classes can define a default response mode for their authorization response by overriding the static *default_response_mode* member.

Custom Validators

class oauthlib.oauth2.rfc6749.grant_types.base.**ValidatorsContainer**(*post_auth*, *post_token*, *pre_auth*, *pre_token*)

Container object for holding custom validator callables to be invoked as part of the grant type *validate_authorization_request()* or *validate_authorization_request()* methods on the various grant types.

Authorization validators must be callables that take a request object and return a dict, which may contain items to be added to the *request_info* returned from the grant_type after validation.

Token validators must be callables that take a request object and return None.

Both authorization validators and token validators may raise OAuth2 exceptions if validation conditions fail.

Authorization validators added to *pre_auth* will be run BEFORE the standard validations (but after the critical ones that raise fatal errors) as part of *validate_authorization_request()*

Authorization validators added to *post_auth* will be run AFTER the standard validations as part of *validate_authorization_request()*

Token validators added to *pre_token* will be run BEFORE the standard validations as part of *validate_token_request()*

Token validators added to *post_token* will be run AFTER the standard validations as part of *validate_token_request()*

For example:

```
>>> def my_auth_validator(request):
...     return {'myval': True}
>>> auth_code_grant = AuthorizationCodeGrant(request_validator)
>>> auth_code_grant.custom_validators.pre_auth.append(my_auth_validator)
>>> def my_token_validator(request):
...     if not request.everything_okay:
...         raise errors.OAuth2Error("uh-oh")
>>> auth_code_grant.custom_validators.post_token.append(my_token_validator)
```

JWT Tokens

Not yet implemented. Track progress in GitHub issue 50.

Grant types are what make OAuth 2 so flexible. The Authorization Code grant is very similar to OAuth 1 (with less crypto), the Implicit grant serves less secure applications such as mobile applications, the Resource Owner Password Credentials grant allows for legacy applications to incrementally transition to OAuth 2, the Client Credentials grant is excellent for embedded services and backend applications.

The main purpose of the grant types is to authorize access to protected resources in various ways with different security credentials.

Naturally, OAuth 2 allows for extension grant types to be defined and OAuthLib attempts to cater for easy inclusion of this as much as possible.

OAuthlib also offers hooks for registering your own custom validations for use with the existing grant type handlers (*oauthlib.oauth2.rfc6749.grant_types.base.ValidatorsContainer*). In some situations, this may be more convenient than subclassing or writing your own extension grant type.

Certain grant types allow the issuing of refresh tokens which will allow a client to request new tokens for as long as you as provider allow them too. In general, OAuth 2 tokens should expire quickly and rather than annoying the user by require them to go through the authorization redirect loop you may use the refresh token to get a new access token. Refresh tokens, contrary to what their name suggest, are components of a grant type rather than token types (like Bearer tokens), much like the authorization code in the authorization code grant.

Tokens

The main token type of OAuth 2 is Bearer tokens and that is what OAuthLib currently supports. Other tokens, such as JWT, SAML and possibly MAC (if the spec matures) can easily be added (and will be in due time).

The purpose of a token is to authorize access to protected resources to a client (i.e. your G+ feed).

The spec requires a `token_type` in access token responses, but some providers, notably Facebook, do not provide this information. Per the robustness principle, we default to the `Bearer` token type if this value is missing. You can force a `MissingTokenTypeError` exception instead, by setting `OAUTHLIB_STRICT_TOKEN_TYPE` in the environment.

Bearer Tokens

The most common OAuth 2 token type. It provides very little in terms of security and relies heavily upon the ability of the client to keep the token secret.

Bearer tokens are the default setting with all configured endpoints. Generally you will not need to ever construct a token yourself as the provided servers will do so for you.

class `oauthlib.oauth2.`**`BearerToken`**(*request_validator=None*, *token_generator=None*, *expires_in=None*, *refresh_token_generator=None*)

`create_token`(*request*, *refresh_token=False*, *save_token=True*)
Create a BearerToken, by default without refresh token.

SAML Tokens

Not yet implemented. Track progress in GitHub issue 49.

MAC tokens

Not yet implemented. Track progress in GitHub issue 29. Might never be supported depending on whether the work on the specification is resumed or not.

CHAPTER 10

OpenID Connect

OpenID Connect represents a substantial set of behaviors and interactions built on the foundations of OAuth2. OAuthLib supports OpenID Connect Authentication flows when the initial grant type request's `scope` parameter contains `openid`. Clients wishing to provide this support must implement several new features within their `RequestValidator` subclass.

ID Tokens

The creation of ID Tokens is ultimately done not by OAuthLib but by your `RequestValidator` subclass. This is because their content is dependent on your implementation of users, their attributes, any claims you may wish to support, as well as the details of how you model the notion of a Client Application. As such OAuthLib simply calls your validator's `get_id_token` method at the appropriate times during the authorization flow, depending on the grant type requested (Authorization Code, Implicit, Hybrid, etc.)

class `oauthlib.oauth2.`**`RequestValidator`**

`get_id_token`(*token*, *token_handler*, *request*)

In the OpenID Connect workflows when an ID Token is requested this method is called. Subclasses should implement the construction, signing and optional encryption of the ID Token as described in the OpenID Connect spec.

In addition to the standard OAuth2 request properties, the request may also contain these OIDC specific properties which are useful to this method:

- nonce, if workflow is implicit or hybrid and it was provided
- claims, if provided to the original Authorization Code request

The token parameter is a dict which may contain an `access_token` entry, in which case the resulting ID Token *should* include a calculated `at_hash` claim.

Similarly, when the request parameter has a `code` property defined, the ID Token *should* include a calculated `c_hash` claim.

http://openid.net/specs/openid-connect-core-1_0.html (sections 3.1.3.6, 3.2.2.10, 3.3.2.11)

Parameters

- **token** – A Bearer token dict
- **token_handler** – the token handler (BearerToken class)
- **request** – the HTTP Request (oauthlib.common.Request)

Returns The ID Token (a JWS signed JWT)

RequestValidator Extensions

Four methods must be implemented in your validator subclass if you wish to support OpenID Connect:

class `oauthlib.oauth2.`**RequestValidator**

get_id_token(*token*, *token_handler*, *request*)

In the OpenID Connect workflows when an ID Token is requested this method is called. Subclasses should implement the construction, signing and optional encryption of the ID Token as described in the OpenID Connect spec.

In addition to the standard OAuth2 request properties, the request may also contain these OIDC specific properties which are useful to this method:

•nonce, if workflow is implicit or hybrid and it was provided

•claims, if provided to the original Authorization Code request

The token parameter is a dict which may contain an `access_token` entry, in which case the resulting ID Token *should* include a calculated `at_hash` claim.

Similarly, when the request parameter has a `code` property defined, the ID Token *should* include a calculated `c_hash` claim.

http://openid.net/specs/openid-connect-core-1_0.html (sections 3.1.3.6, 3.2.2.10, 3.3.2.11)

Parameters

- **token** – A Bearer token dict
- **token_handler** – the token handler (BearerToken class)
- **request** – the HTTP Request (oauthlib.common.Request)

Returns The ID Token (a JWS signed JWT)

validate_silent_authorization(*request*)

Ensure the logged in user has authorized silent OpenID authorization.

Silent OpenID authorization allows access tokens and id tokens to be granted to clients without any user prompt or interaction.

Parameters **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

validate_silent_login(*request*)
Ensure session user has authorized silent OpenID login.

If no user is logged in or has not authorized silent login, this method should return False.

If the user is logged in but associated with multiple accounts and not selected which one to link to the token then this method should raise an oauthlib.oauth2.AccountSelectionRequired error.

Parameters **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

validate_user_match(*id_token_hint*, *scopes*, *claims*, *request*)
Ensure client supplied user id hint matches session user.

If the sub claim or id_token_hint is supplied then the session user must match the given ID.

Parameters

- **id_token_hint** – User identifier string.
- **scopes** – List of OAuth 2 scopes and OpenID claims (strings).
- **claims** – OpenID Connect claims dict.
- **request** – The HTTP Request (oauthlib.common.Request)

Return type True or False

Method is used by:

- OpenIDConnectAuthCode
- OpenIDConnectImplicit
- OpenIDConnectHybrid

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

R

S

T

V

W