sanction Documentation

Release 0.4

Demian Brecht

May 14, 2014

Contents

sanction [sangk-shuhn]: authoritative permission or approval, as for an action.

Contents

CHAPTER 1

Overview

sanction is a lightweight, dead simple client implementation of the OAuth2 protocol. The major goals of the library are:

- Grok me
  - Variations on OAuth2 client implementation range from a few hundred LOC to thousands. In a Pythonic world, there's absolutely no need for this when simply dealing with the client side of the spec. Currently, sanction sits at a whopping 65 LOC, one class. This makes the library tremendously easy to grok.
- Support multiple providers
  - Most providers have varying levels of diversion from the official spec. The goal with this library is to either handle these diversions natively, or expose a method to allow client code to deal with it efficiently and effectively.
- Support all server-side OAuth2 flows
  - Three of the four OAuth2 flows should be supported by this library. Currently, only authorization code and client credential flows have been tested due to lack of other (known) implementations.

sanction has been tested with the following OAuth2 providers:

- Facebook (include the test API)
- Google
- Foursquare
- bitly
- GitHub
- StackExchange
- Instagram
- DeviantArt

note The intention of the sanction library is to not only provide accessibility to the OAuth2 authorization code flow, but all server-side flows. However, due to lack of implementations, the only tested currently teseted flows are authorization code and client credentials.

CHAPTER 2

Quickstart

For the quickstart, authorization code grant flow is assumed, as is the Bearer token type. If you're unfamiliar with these terms, chances are that they're what you're looking for as it's the default in most public OAuth2 provider implementations (Google, Facebook, Foursquare, etc.).

Introducing this library should be rather trivial (in the usual basic case). There are three steps required in the most common use case (Google is assumed to be the provider throughout sample code):

You can also take a look at the example code in `/example`.

2.1 Instantiation

To access protected resources via the OAuth2 protocol, you must instantiate a `Client` and pass it relevant endpoints for your current operation:

```
from sanction.client import Client

# instantiating a client to get the auth URI
c = Client(auth_endpoint="https://accounts.google.com/o/oauth2/auth",
    client_id=config["google.client_id"],
    redirect_uri="http://localhost:8080/login/google")

# instantiating a client to process OAuth2 response
c = Client(token_endpoint="https://accounts.google.com/o/oauth2/token",
    resource_endpoint="https://www.googleapis.com/oauth2/v1",
    redirect_uri="http://localhost:8080/login/google",
    client_id=config["google.client_id"],
    client_secret=config["google.client_secret"])
```

Of course, you may create the config `dict` in your preferred method, the above is simply for demonstration using the required config settings (the example project uses `ConfigParser` against an `.ini` file for settings.

2.2 Authorization Request

The next step is to redirect the user agent to the provider's authentication/ authorization uri (continuation from previous code block):

```
scope_req = 'scope1,scope2'
my_redirect(c.auth_uri(scope_req))
```

note (New in 0.4) The scope is assumed to be consistent with the expectations of the provider. For example, scope being passed to Facebook should be a single string with comma-delimited entities while Google should be delimited by spaces (i.e. Facebook: `scope_req = 'scope1,scope2'`).

You can also elect to use the optional `state` parameter to pass a CSRF token that will be included if the provider's response:

```
my_redirect(c.auth_uri(state=my_state))
```

note It is **strongly** encouraged that you use the `state` parameter to offer CSRF protection. It is also up to you to process the `state` parameter and handle redirection accordingly *before* calling `request_token`.

2.3 Access Token Request

When the user has granted or denied resource access to your application, they will be redirected to the `redirect_uri` as specified by the value of the `GET` param. In order to request an access token from the provider, you must request an access token from the provider:

```
c.request_token(response_dict)
```

The default parser (`sanction._default_parser`) will attempt to decode JSON, followed by an attempt to parse querystring-formatted data (i.e. access token data returned by Facebook). If the provider you're working with has a different requirement, you can use the `parser` parameter of `request_token` to pass another callable that will be responsible for decoding the returned data.

2.4 Refreshing Access Tokens

Some (not all) providers provide the ability to refresh a given access token, giving an application to users' data even if they're offline (Google is one of these providers). If your client previously received a refresh token with the initial code/token exchange, then you can use the `request_token` API to request a refreshed token:

```
c.request_token(grant_type='refresh_token',
    refresh_token=my_refresh_token)
```

An example of this is shown in the Google login handler in the sample app.

2.5 Resource Request

If the user has granted access and your config settings are correct, you should then be able to access protected resources through the adapter's API:

```
c.request("/userinfo")
```

If the provider has deviated from the OAuth2 spec and the response isn't JSON (i.e. Stack Exchange), you can pass a custom parser to `request`:

```
c.request("/userinfo", parser=lambda c: dosomething(c))
```

2.6 API

class sanction.**Client**(*auth_endpoint=None*, *token_endpoint=None*, *resource_endpoint=None*, *client_id=None*, *client_secret=None*, *token_transport=None*)

OAuth 2.0 client object

auth_uri(*redirect_uri=None*, *scope=None*, *scope_delim=None*, *state=None*, ***kwargs*)

Builds the auth URI for the authorization endpoint

Parameters

- **scope** – (optional) The *scope* parameter to pass for authorization. The format should match that expected by the provider (i.e. Facebook expects comma-delimited, while Google expects space-delimited)
- **state** – (optional) The *state* parameter to pass for authorization. If the provider follows the OAuth 2.0 spec, this will be returned to your *redirect_uri* after authorization. Generally used for CSRF protection.
- ****kwargs** – Any other querystring parameters to be passed to the provider.

request(*url*, *method=None*, *data=None*, *headers=None*, *parser=None*)

Request user data from the resource endpoint :param url: The path to the resource and querystring if required :param method: HTTP method. Defaults to `GET` unless data is not None

in which case it defaults to `POST`

Parameters

- **data** – Data to be POSTed to the resource endpoint
- **parser** – Parser callback to deal with the returned data. Defaults to ‘‘json.loads‘.‘

request_token(*parser=None*, *redirect_uri=None*, ***kwargs*)

Request an access token from the token endpoint. This is largely a helper method and expects the client code to understand what the server expects. Anything that’s passed into `**kwargs` will be sent (‘‘urlencode‘‘d) to the endpoint. Client secret and client ID are automatically included, so are not required as kwargs. For example:

```
# if requesting access token from auth flow:
{
    'code': rval_from_auth,
}

# if refreshing access token:
{
    'refresh_token': stored_refresh_token,
    'grant_type': 'refresh_token',
}
```

Parameters **parser** – Callback to deal with returned data. Not all providers use JSON.

CHAPTER 3

Notes

There are no implementations for individual OAuth2-exposed resources. This is not the intention of the library and will not be added.

CHAPTER 4

Thanks

- Alec Taylor: Example code refactor.
- Jake Basile: Pointing out the oversight of the `refresh_token`.
- Ricky Elrod: Python 2.6 string index fix.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

S