django-oauth2-provider Documentation

Release 0.2.4

Alen Mujezinovic

March 01, 2013

CONTENTS

django-oauth2-provider is a Django application that provides customizable OAuth2 authentication for your Django projects.

The default implementation makes reasonable assumptions about the allowed grant types and provides clients with two easy accessible URL endpoints. (`provider.oauth2.urls`)

If you require custom database backends, URLs, wish to extend the OAuth2 protocol as defined in Section 8 or anything else, you can override the default behaviours by subclassing the views in `provider.views` and add your specific use cases.

CHAPTER

ONE

GETTING STARTED

1.1 Getting started

1.1.1 Installation

```
$ pip install django-oauth2-provider
```

1.1.2 Configuration

Add OAuth2 Provider to `INSTALLED_APPS`

```
INSTALLED_APPS = (
    # ...
    'provider',
    'provider.oauth2',
)
```

Modify your settings to match your needs

The default settings are available in `provider.constants`.

Include the OAuth 2 views

Add `provider.oauth2.urls` to your root `urls.py` file.

```
url(r'^oauth2/', include('provider.oauth2.urls', namespace = 'oauth2')),
```

Note: The namespace argument is required.

Sync your database

```
$ python manage.py syncdb
$ python manage.py migrate
```

1.1.3 How to request an `access token` for the first time ?

Create a `client` entry in your database

Note: To find out which type of `client` you need to create, read Section 2.1.

To create a new entry, simply use the django admin panel.

Request an access token

Your client interface – I mean by that your iOS code, HTML code, or whatever else language – just have to submit a `POST` request at the url `/oauth2/access_token` with the following fields :

- client_id the client id you've just configured at the previous step.
- client_secret again configured at the previous step.
- username the username with which you want to log in.
- password well, that speaks for itself.

This is only one way to authenticate with OAuth 2, there is other methods but I will only show you the `PasswordGrant` type one in this quick "Getting started" guide.

Note: Remember that you SHOULD always use HTTPS for all your OAuth 2 requests otherwise you won't be secured.

Now you can use the command line to check that your local configuration is working :

```
$ curl -X POST -d "client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET&grant_type=password&use
```

Here is the response you should get :

```
{"access_token": "<your-access-token>", "scope": "read", "expires_in": 86399, "refresh_token": "<you
```

CHAPTER

TWO

API

2.1 *provider*

2.1.1 *provider.constants*

provider.constants.**RESPONSE_TYPE_CHOICES**

Settings *OAUTH_RESPONSE_TYPE_CHOICES*

The response types as outlined by Section 3.1.1

provider.constants.**SCOPES**

Settings *OAUTH_SCOPES*

A choice of scopes. A detailed implementation is left to the developer. The current default implementation in `provider.oauth2.scope` makes use of bit shifting operations to combine read and write permissions.

provider.constants.**EXPIRE_DELTA**

Settings *OAUTH_EXPIRE_DELTA*

Default *datetime.timedelta(days=365)*

The time to expiry for access tokens as outlined in Section 4.2.2 and Section 5.1.

provider.constants.**EXPIRE_CODE_DELTA**

Settings *OAUTH_EXPIRE_CODE_DELTA*

Default *datetime.timedelta(seconds=10*60)*

The time to expiry for an authorization code grant as outlined in Section 4.1.2.

provider.constants.**ENFORCE_SECURE**

Settings *OAUTH_ENFORCE_SECURE*

Default *False*

To enforce secure communication on application level, set to *True*.

provider.constants.**SESSION_KEY**

Settings *OAUTH_SESSION_KEY*

Default *"oauth"*

Session key prefix to store temporary data while the user is completing the authentication / authorization process.

2.1.2 *provider.forms*

class `provider.forms.`**`OAuthForm`**(**args*, ***kwargs*)

Form class that creates shallow error dicts and exists early when a `OAuthValidationError` is raised.

The shallow error dict is reused when returning error responses to the client.

The different types of errors are outlined in Section 4.2.2.1 and Section 5.2.

exception `provider.forms.`**`OAuthValidationError`**

Exception to throw inside `OAuthForm` if any OAuth2 related errors are encountered such as invalid grant type, invalid client, etc.

`OAuthValidationError` expects a dictionary outlining the OAuth error as its first argument when instantiating.

Example

```
class GrantValidationForm(OAuthForm):
    grant_type = forms.CharField()

    def clean_grant(self):
        if not self.cleaned_data.get('grant_type') == 'code':
            raise OAuthValidationError({
                'error': 'invalid_grant',
                'error_description': "%s is not a valid grant type" % (
                    self.cleaned_data.get('grant_type'))
            })
```

The different types of errors are outlined in Section 4.2.2.1 and Section 5.2.

2.1.3 *provider.scope*

Default scope implementation relying on bit shifting. See `provider.constants.SCOPES` for the list of available scopes.

Scopes can be combined, such as `"read write"`. Note that a single `"write"` scope is *not* the same as `"read write"`.

See `provider.scope.to_int` on how scopes are combined.

`provider.scope.`**`check`**(*wants*, *has*)

Check if a desired scope `wants` is part of an available scope `has`.

Returns `False` if not, return `True` if yes.

Example

If a list of scopes such as

```
READ = 1 << 1
WRITE = 1 << 2
READ_WRITE = READ | WRITE

SCOPES = (
    (READ, 'read'),
    (WRITE, 'write'),
    (READ_WRITE, 'read+write'),
)
```

is defined, we can check if a given scope is part of another:

```
>>> from provider import scope
>>> scope.check(READ, READ)
True
>>> scope.check(WRITE, READ)
False
>>> scope.check(WRITE, WRITE)
True
>>> scope.check(READ, WRITE)
False
>>> scope.check(READ, READ_WRITE)
True
>>> scope.check(WRITE, READ_WRITE)
True
```

provider.scope.**names**(*scope*)
Returns a list of scope names as defined in provider.constants.SCOPES for a given scope integer.

```
>>> assert ['read', 'write'] == provider.scope.names(provider.constants.READ_WRITE)
```

provider.scope.**to_int**(**names*, ***kwargs*)
Turns a list of scope names into an integer value.

```
>>> scope.to_int('read')
2
>>> scope.to_int('write')
6
>>> scope.to_int('read', 'write')
6
>>> scope.to_int('invalid')
0
>>> scope.to_int('invalid', default = 1)
1
```

provider.scope.**to_names**(*scope*)
Returns a list of scope names as defined in provider.constants.SCOPES for a given scope integer.

```
>>> assert ['read', 'write'] == provider.scope.names(provider.constants.READ_WRITE)
```

2.1.4 *provider.templatetags.scope*

provider.templatetags.scope.**scopes**(*scope_int*)
Wrapper around provider.scope.names to turn an int into a list of scope names in templates.

2.1.5 *provider.utils*

provider.utils.**get_code_expiry**()
Return a datetime object indicating when an authorization code should expire. Can be customized by setting settings.OAUTH_EXPIRE_CODE_DELTA to a datetime.timedelta object.

provider.utils.**get_token_expiry**()
Return a datetime object indicating when an access token should expire. Can be customized by setting settings.OAUTH_EXPIRE_DELTA to a datetime.timedelta object.

provider.utils.**long_token**()
Generate a hash that can be used as an application secret

provider.utils.**short_token**()
Generate a hash that can be used as an application identifier

2.1.6 *provider.views*

class provider.views.**AccessToken**(***kwargs*)
AccessToken handles creation and refreshing of access tokens.

Implementations must implement a number of methods:

- get_authorization_code_grant
- get_refresh_token_grant
- get_password_grant
- create_access_token
- create_refresh_token
- invalidate_grant
- invalidate_access_token
- invalidate_refresh_token

The default implementation supports the grant types defined in grant_types.

According to Section 4.4.2 this endpoint too must support secure communication. For strict enforcement of secure communication at application level set settings.OAUTH_ENFORCE_SECURE to True.

According to Section 3.2 we can only accept POST requests.

Returns with a status code of *400* in case of errors. *200* in case of success.

access_token_response(*access_token*)
Returns a successful response after creating the access token as defined in Section 5.1.

authentication = ()
Authentication backends used to authenticate a particular client.

authorization_code(*request*, *data*, *client*)
Handle grant_type=authorization_code requests as defined in Section 4.1.3.

create_access_token(*request*, *user*, *scope*, *client*)
Override to handle access token creation.

Returns object - Access token

create_refresh_token(*request*, *user*, *scope*, *access_token*, *client*)
Override to handle refresh token creation.

Returns object - Refresh token

error_response(*error*, *mimetype='application/json'*, *status=400*, ***kwargs*)
Return an error response to the client with default status code of *400* stating the error as outlined in Section 5.2.

get(*request*)
As per Section 3.2 the token endpoint *only* supports POST requests. Returns an error response.

get_authorization_code_grant(*request*, *data*, *client*)
Return the grant associated with this request or an error dict.

Returns tuple - (True or False, grant or error_dict)

get_handler(*grant_type*)
Return a function or method that is capable handling the `grant_type` requested by the client or return `None` to indicate that this type of grant type is not supported, resulting in an error response.

get_password_grant(*request*, *data*, *client*)
Return a user associated with this request or an error dict.

Returns `tuple - (True or False, user or error_dict)`

get_refresh_token_grant(*request*, *data*, *client*)
Return the refresh token associated with this request or an error dict.

Returns `tuple - (True or False, token or error_dict)`

grant_types = ['authorization_code', 'refresh_token', 'password']
The default grant types supported by this view.

invalidate_access_token(*access_token*)
Override to handle access token invalidation. When a new access token is created from a refresh token, the old one is *always* invalidated.

Return None

invalidate_grant(*grant*)
Override to handle grant invalidation. A grant is invalidated right after creating an access token from it.

Return None

invalidate_refresh_token(*refresh_token*)
Override to handle refresh token invalidation. When requesting a new access token from a refresh token, the old one is *always* invalidated.

Return None

password(*request*, *data*, *client*)
Handle `grant_type=password` requests as defined in Section 4.3.

post(*request*)
As per Section 3.2 the token endpoint *only* supports POST requests.

refresh_token(*request*, *data*, *client*)
Handle `grant_type=refresh_token` requests as defined in Section 6.

class `provider.views.`**Authorize**(***kwargs*)
View to handle the client authorization as outlined in Section 4. Implementation must override a set of methods:

- `get_redirect_url`
- `get_request_form`
- `get_authorization_form`
- `get_client`
- `save_authorization`

`Authorize` renders the `provider/authorize.html` template to display the authorization form.

On successful authorization, it redirects the user back to the defined client callback as defined in Section 4.1.2.

On authorization fail `Authorize` displays an error message to the user with a modified redirect URL to the callback including the error and possibly description of the error as defined in Section 4.1.2.1.

error_response(*request*, *error*, ***kwargs*)
Return an error to be displayed to the resource owner if anything goes awry. Errors can include invalid

clients, authorization denials and other edge cases such as a wrong `redirect_uri` in the authorization request.

Parameters

- **request** – `django.http.HttpRequest`
- **error** – `dict` The different types of errors are outlined in Section 4.2.2.1

get_authorization_form(*request*, *client*, *data*, *client_data*)
Return a form that is capable of authorizing the client to the resource owner.

Returns `django.forms.Form`

get_client(*client_id*)
Return a client object from a given client identifier. Return `None` if no client is found. An error will be displayed to the resource owner and presented to the client upon the final redirect.

get_redirect_url(*request*)

Returns `str` - The client URL to display in the template after authorization succeeded or failed.

get_request_form(*client*, *data*)
Return a form that is capable of validating the request data captured by the `Capture` view. The form must accept a keyword argument `client`.

save_authorization(*request*, *client*, *form*, *client_data*)
Save the authorization that the user granted to the client, involving the creation of a time limited authorization code as outlined in Section 4.1.2.

Should return `None` in case authorization is not granted. Should return a string representing the authorization code grant.

Returns `None`, `str`

class `provider.views.`**Capture**(***kwargs*)
As stated in section Section 3.1.2.5 this view captures all the request parameters and redirects to another URL to avoid any leakage of request parameters to potentially harmful JavaScripts.

This application assumes that whatever web-server is used as front-end will handle SSL transport.

If you want strict enforcement of secure communication at application level, set `settings.OAUTH_ENFORCE_SECURE` to `True`.

The actual implementation is required to override `get_redirect_url()`.

get_redirect_url(*request*)
Return a redirect to a URL where the resource owner (see Section 1) authorizes the client (also Section 1).

Returns `django.http.HttpResponseRedirect`

class `provider.views.`**Mixin**
Mixin providing common methods required in the OAuth view defined in `provider.views`.

authenticate(*request*)
Authenticate a client against all the backends configured in `authentication`.

cache_data(*request*, *data*, *key='params'*)
Cache data in the session store.

Parameters

- **request** – `django.http.HttpRequest`
- **data** – Arbitrary data to store.

- **key** – *str* The key under which to store the data.

clear_data(*request*)
Clear all OAuth related data from the session store.

get_data(*request*, *key='params'*)
Return stored data from the session store.

Parameters **key** – *str* The key under which the data was stored.

exception provider.views.**OAuthError**
Exception to throw inside any views defined in provider.views.

Any OAuthError thrown will be signalled to the API consumer.

OAuthError expects a dictionary as its first argument outlining the type of error that occured.

Example

```
raise OAuthError({'error': 'invalid_request'})
```

The different types of errors are outlined in Section 4.2.2.1 and Section 5.2.

class provider.views.**OAuthView**(***kwargs*)
Base class for any view dealing with the OAuth flow. This class overrides the dispatch method of TemplateView to add no-caching headers to every response as outlined in Section 5.1.

class provider.views.**Redirect**(***kwargs*)
Redirect the user back to the client with the right query parameters set. This can be either parameters indicating success or parameters indicating an error.

2.2 provider.oauth2

2.2.1 provider.oauth2.forms

class provider.oauth2.forms.**AuthorizationCodeGrantForm**(**args*, ***kwargs*)
Check and return an authorization grant.

clean()
Make sure that the scope is less or equal to the scope allowed on the grant!

class provider.oauth2.forms.**AuthorizationForm**(**args*, ***kwargs*)
A form used to ask the resource owner for authorization of a given client.

class provider.oauth2.forms.**AuthorizationRequestForm**(**args*, ***kwargs*)
This form is used to validate the request data that the authorization endpoint receives from clients.

Included data is specified in Section 4.1.1.

clean_redirect_uri()

Section 3.1.2 **The redirect value has to match what was saved on the** authorization server.

clean_response_type()
Section 3.1.1 Lists of values are space delimited.

redirect_uri = None
Where the client would like to redirect the user back to. This has to match whatever value was saved while creating the client.

response_type = None
"code" or "token" depending on the grant type.

scope = None
The scope that the authorization should include.

state = None
Opaque - just pass back to the client for validation.

class provider.oauth2.forms.**ClientAuthForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=u':'*, *empty_permitted=False*)
Client authentication form. Required to make sure that we're dealing with a real client. Form is used in provider.oauth2.backends to validate the client.

class provider.oauth2.forms.**ClientForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=u':'*, *empty_permitted=False*, *instance=None*)
Form to create new consumers.

class provider.oauth2.forms.**PasswordGrantForm**(**args*, ***kwargs*)
Validate the password of a user on a password grant request.

class provider.oauth2.forms.**RefreshTokenGrantForm**(**args*, ***kwargs*)
Checks and returns a refresh token.

clean()
Make sure that the scope is less or equal to the previous scope!

class provider.oauth2.forms.**ScopeChoiceField**(*choices=()*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, **args*, ***kwargs*)
Custom form field that seperates values on space as defined in Section 3.3.

validate(*value*)
Validates that the input is a list or tuple.

class provider.oauth2.forms.**ScopeMixin**
Form mixin to clean scope fields.

clean_scope()
The scope is assembled by combining all the set flags into a single integer value which we can later check again for set bits.

If *no* scope is set, we return the default scope which is the first defined scope in provider.constants.SCOPES.

2.2.2 *provider.oauth2.models*

Default model implementations. Custom database or OAuth backends need to implement these models with fields and and methods to be compatible with the views in provider.views.

class provider.oauth2.models.**AccessToken**(**args*, ***kwargs*)
Default access token implementation. An access token is a time limited token to access a user's resources.

Access tokens are outlined Section 5.

Expected fields:

- user
- token

- client - Client
- expires - datetime.datetime
- scope

Expected methods:

- **get_expire_delta() - returns an integer representing seconds to** expiry

get_expire_delta()
Return the number of seconds until this token expires.

class provider.oauth2.models.**Client**(**args*, ***kwargs*)
Default client implementation.

Expected fields:

- user
- name
- url
- redirect_url
- client_id
- client_secret
- client_type

Clients are outlined in the Section 2 and its subsections.

class provider.oauth2.models.**Grant**(**args*, ***kwargs*)
Default grant implementation. A grant is a code that can be swapped for an access token. Grants have a limited lifetime as defined by provider.constants.EXPIRE_CODE_DELTA and outlined in Section 4.1.2

Expected fields:

- user
- client - Client
- code
- expires - datetime.datetime
- redirect_uri
- scope

class provider.oauth2.models.**RefreshToken**(**args*, ***kwargs*)
Default refresh token implementation. A refresh token can be swapped for a new access token when said token expires.

Expected fields:

- user
- token
- access_token - AccessToken
- client - Client
- expired - boolean

2.2.3 *provider.oauth2.urls*

The default implementation of the OAuth provider includes two public endpoints that are meant for client (as defined in Section 1) interaction.

^authorize/$
This is the URL where a client should redirect a user to for authorization.

This endpoint expects the parameters defined in Section 4.1.1 and returns responses as defined in Section 4.1.2 and Section 4.1.2.1.

^access_token/$
This is the URL where a client exchanges a grant for an access tokens.

This endpoint expects different parameters depending on the grant type:

- Access tokens: Section 4.1.3
- Refresh tokens: Section 6
- Password grant: Section 4.3.2

This endpoint returns responses depending on the grant type:

- Access tokens: Section 4.1.4 and Section 5.1
- Refresh tokens: Section 4.1.4 and Section 5.1
- Password grant: Section 5.1

To override, remove or add grant types, override the appropriate methods on `provider.views.AccessToken` and / or `provider.oauth2.views.AccessTokenView`.

Errors are outlined in Section 5.2.

2.2.4 *provider.oauth2.views*

class `provider.oauth2.views.`**`AccessTokenView`**(***kwargs*)
Implementation of `provider.views.AccessToken`.

Note: This implementation does provide all default grant types defined in `provider.views.AccessToken.grant_types`. If you wish to disable any, you can override the `get_handler()` method *or* the `grant_types` list.

---

class `provider.oauth2.views.`**`Authorize`**(***kwargs*)
Implementation of `provider.views.Authorize`.

class `provider.oauth2.views.`**`Capture`**(***kwargs*)
Implementation of `provider.views.Capture`.

class `provider.oauth2.views.`**`Redirect`**(***kwargs*)
Implementation of `provider.views.Redirect`

CHAPTER

THREE

CHANGES

3.1 v 0.2

- *Breaking change* Moved `provider.oauth2.scope` to `provider.scope`
- *Breaking change* Replaced the write scope with a new write scope that includes reading
- Default scope for new `provider.oauth2.models.AccessToken` is now `provider.constants.SCOPES[0][0]`
- Access token response returns a space seperated list of scopes instead of an integer value

Made by Caffeinehit.

PYTHON MODULE INDEX

p

provider.constants, **??**
provider.forms, **??**
provider.oauth2.forms, **??**
provider.oauth2.models, **??**
provider.oauth2.urls, **??**
provider.oauth2.views, **??**
provider.scope, **??**
provider.templatetags.scope, **??**
provider.utils, **??**
provider.views, **??**