django-oauth2-provider Documentation

Release 0.2.7-dev

Alen Mujezinovic

Aug 16, 2017

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 1

Getting started

Getting started

Installation

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

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
```

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

Assuming that you've used the same URL configuration as above, your client needs to submit a `POST` request to `/oauth2/access_token` including the following parameters:

- `client_id` - The client ID you've configured in the Django admin.
- `client_secret` - The client secret configured in the Django admin.
- `username` - The username with which you want to log in.
- `password` - The password corresponding to the user you're logging in with.

Request

```
$ curl -X POST -d "client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET&grant_
↪type=password&username=YOUR_USERNAME&password=YOUR_PASSWORD" http://localhost:8000/
↪oauth2/access_token/
```

Response

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

This particular way of obtaining an access token is called a **Password Grant**. All the other ways of acquiring an access token are outlined in Section 4.

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

CHAPTER 2

API

provider

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.`**`DELETE_EXPIRED`**

Settings *OAUTH_DELETE_EXPIRED*

Default *False*

To remove expired tokens immediately instead of letting them persist, set to *True*.

`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.

`provider.constants.`**`SINGLE_ACCESS_TOKEN`**

Settings *OAUTH_SINGLE_ACCESS_TOKEN*

Default *False*

To have the provider only create and retrieve one access token per user/client/scope combination, set to *True*.

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.

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)
```

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.

provider.utils

`provider.utils.`**`deserialize_instance`**`(`*model*, *data={}*`)`
Translate raw data into a model instance.

`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`**`(`*public=True*`)`
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.`**`serialize_instance`**`(`*instance*`)`
Since Django 1.6 items added to the session are no longer pickled, but JSON encoded by default. We are storing partially complete models in the session (user, account, token, ...). We cannot use standard Django serialization, as these are models are not "complete" yet. Serialization will start complaining about missing relations et al.

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

provider.views

provider.oauth2

provider.oauth2.forms

provider.oauth2.models

provider.oauth2.urls

provider.oauth2.views

CHAPTER 3

Changes

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

Index