Django OAuth Toolkit Documentation

Release 0.11.0

Evonove

August 09, 2017

Contents

Django OAuth Toolkit can help you providing out of the box all the endpoints, data and logic needed to add OAuth2 capabilities to your Django projects. Django OAuth Toolkit makes extensive use of the excellent OAuthLib, so that everything is rfc-compliant.

See our Changelog for information on updates.

CHAPTER 1

Support

If you need support please send a message to the Django OAuth Toolkit Google Group

CHAPTER 2

Requirements

- Python 2.7, 3.2, 3.3, 3.4, 3.5
- Django 1.7, 1.8, 1.9

CHAPTER 3

Index

Installation

Install with pip

    pip install django-oauth-toolkit

Add *oauth2_provider* to your *INSTALLED_APPS*

```
INSTALLED_APPS = (
    ...
    'oauth2_provider',
)
```

If you need an OAuth2 provider you'll want to add the following to your urls.py

```
urlpatterns = [
    ...
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
]
```

Sync your database

```
$ python manage.py migrate oauth2_provider
```

Next step is our first tutorial.

Tutorials

Part 1 - Make a Provider in a Minute

Scenario

You want to make your own *Authorization Server* to issue access tokens to client applications for a certain API.

Start Your App

During this tutorial you will make an XHR POST from a Heroku deployed app to your localhost instance. Since the domain that will originate the request (the app on Heroku) is different from the destination domain (your local instance), you will need to install the django-cors-middleware app. These "cross-domain" requests are by default forbidden by web browsers unless you use CORS.

Create a virtualenv and install *django-oauth-toolkit* and *django-cors-middleware*:

```
pip install django-oauth-toolkit django-cors-middleware
```

Start a Django project, add *oauth2_provider* and *corsheaders* to the installed apps, and enable admin:

```
INSTALLED_APPS = {
    'django.contrib.admin',
    # ...
    'oauth2_provider',
    'corsheaders',
}
```

Include the Django OAuth Toolkit urls in your *urls.py*, choosing the urlspace you prefer. For example:

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
    # ...
]
```

Include the CORS middleware in your *settings.py*:

```
MIDDLEWARE_CLASSES = (
    # ...
    'corsheaders.middleware.CorsMiddleware',
    # ...
)
```

Allow CORS requests from all domains (just for the scope of this tutorial):

```
CORS_ORIGIN_ALLOW_ALL = True
```

Include the required hidden input in your login template, *registration/login.html*. The `{{ next }}` template context variable will be populated with the correct redirect value. See the Django documentation for details on using login templates.

```
<input type="hidden" name="next" value="{{ next }}" />
```

As a final step, execute the migrate command, start the internal server, and login with your credentials.

Create an OAuth2 Client Application

Before your *Application* can use the *Authorization Server* for user login, you must first register the app (also known as the *Client*.) Once registered, your app will be granted access to the API, subject to approval by its users.

Let's register your application.

Point your browser to http://localhost:8000/o/applications/ and add an Application instance. *Client id* and *Client Secret* are automatically generated; you have to provide the rest of the informations:

- *User*: the owner of the Application (e.g. a developer, or the currently logged in user.)

- *Redirect uris*: Applications must register at least one redirection endpoint before using the authorization endpoint. The *Authorization Server* will deliver the access token to the client only if the client specifies one of the verified redirection uris. For this tutorial, paste verbatim the value *http://django-oauth-toolkit.herokuapp.com/consumer/exchange/*
- *Client type*: this value affects the security level at which some communications between the client application and the authorization server are performed. For this tutorial choose *Confidential*.
- *Authorization grant type*: choose *Authorization code*
- *Name*: this is the name of the client application on the server, and will be displayed on the authorization request page, where users can allow/deny access to their data.

Take note of the *Client id* and the *Client Secret* then logout (this is needed only for testing the authorization process we'll explain shortly)

Test Your Authorization Server

Your authorization server is ready and can begin issuing access tokens. To test the process you need an OAuth2 consumer; if you are familiar enough with OAuth2, you can use curl, requests, or anything that speaks http. For the rest of us, there is a consumer service deployed on Heroku to test your provider.

Build an Authorization Link for Your Users

Authorizing an application to access OAuth2 protected data in an *Authorization Code* flow is always initiated by the user. Your application can prompt users to click a special link to start the process. Go to the Consumer page and complete the form by filling in your application's details obtained from the steps in this tutorial. Submit the form, and you'll receive a link your users can use to access the authorization page.

Authorize the Application

When a user clicks the link, she is redirected to your (possibly local) *Authorization Server*. If you're not logged in, you will be prompted for username and password. This is because the authorization page is login protected by django-oauth-toolkit. Login, then you should see the (not so cute) form a user can use to give her authorization to the client application. Flag the *Allow* checkbox and click *Authorize*, you will be redirected again to the consumer service. If you are not redirected to the correct page after logging in successfully, you probably need to *setup your login template correctly*.

Exchange the token

At this point your authorization server redirected the user to a special page on the consumer passing in an *Authorization Code*, a special token the consumer will use to obtain the final access token. This operation is usually done automatically by the client application during the request/response cycle, but we cannot make a POST request from Heroku to your localhost, so we proceed manually with this step. Fill the form with the missing data and click *Submit*. If everything is ok, you will be routed to another page showing your access token, the token type, its lifetime and the *Refresh Token*.

Refresh the token

The page showing the access token retrieved from the *Authorization Server* also let you make a POST request to the server itself to swap the refresh token for another, brand new access token. Just fill in the missing form fields and click the Refresh button: if everything goes smoothly you will see the access and refresh token change their values,

otherwise you will likely see an error message. When you have finished playing with your authorization server, take note of both the access and refresh tokens, we will use them for the next part of the tutorial.

So let's make an API and protect it with your OAuth2 tokens in the part 2 of the tutorial.

Part 2 - protect your APIs

Scenario

It's very common for an *Authorization Server* being also the *Resource Server*, usually exposing an API to let others access its own resources. Django OAuth Toolkit implements an easy way to protect the views of a Django application with OAuth2, in this tutorial we will see how to do it.

Make your API

We start where we left the part 1 of the tutorial: you have an authorization server and we want it to provide an API to access some kind of resources. We don't need an actual resource, so we will simply expose an endpoint protected with OAuth2: let's do it in a *class based view* fashion!

Django OAuth Toolkit provides a set of generic class based view you can use to add OAuth behaviour to your views. Open your *views.py* module and import the view:

```
from oauth2_provider.views.generic import ProtectedResourceView
from django.http import HttpResponse
```

Then create the view which will respond to the API endpoint:

```
class ApiEndpoint(ProtectedResourceView):
    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, OAuth2!')
```

That's it, our API will expose only one method, responding to *GET* requests. Now open your *urls.py* and specify the URL this view will respond to:

```
from django.conf.urls import url
import oauth2_provider.views as oauth2_views
from django.conf import settings
from .views import ApiEndpoint

# OAuth2 provider endpoints
oauth2_endpoint_views = [
    url(r'^authorize/$', oauth2_views.AuthorizationView.as_view(), name="authorize"),
    url(r'^token/$', oauth2_views.TokenView.as_view(), name="token"),
    url(r'^revoke-token/$', oauth2_views.RevokeTokenView.as_view(), name="revoke-token"),
]

if settings.DEBUG:
    # OAuth2 Application Management endpoints
    oauth2_endpoint_views += [
        url(r'^applications/$', oauth2_views.ApplicationList.as_view(), name="list"),
        url(r'^applications/register/$', oauth2_views.ApplicationRegistration.as_view(), name="regist
        url(r'^applications/(?P<pk>\d+)/$', oauth2_views.ApplicationDetail.as_view(), name="detail"),
        url(r'^applications/(?P<pk>\d+)/delete/$', oauth2_views.ApplicationDelete.as_view(), name="de
        url(r'^applications/(?P<pk>\d+)/update/$', oauth2_views.ApplicationUpdate.as_view(), name="up
    ]

    # OAuth2 Token Management endpoints
```

```
    oauth2_endpoint_views += [
        url(r'^authorized-tokens/$', oauth2_views.AuthorizedTokensListView.as_view(), name="authorize
        url(r'^authorized-tokens/(?P<pk>\d+)/delete/$', oauth2_views.AuthorizedTokenDeleteView.as_vie
            name="authorized-token-delete"),
    ]

urlpatterns = [
    # OAuth 2 endpoints:
    url(r'^o/', include(oauth2_endpoint_views, namespace="oauth2_provider")),

    url(r'^admin/', include(admin.site.urls)),
    url(r'^api/hello', ApiEndpoint.as_view()),  # an example resource endpoint
]
```

You will probably want to write your own application views to deal with permissions and access control but the ones packaged with the library can get you started when developing the app.

Since we inherit from *ProtectedResourceView*, we're done and our API is OAuth2 protected - for the sake of the lazy programmer.

Testing your API

Time to make requests to your API.

For a quick test, try accessing your app at the url */api/hello* with your browser and verify that it responds with a *403* (in fact no *HTTP_AUTHORIZATION* header was provided). You can test your API with anything that can perform HTTP requests, but for this tutorial you can use the online consumer client. Just fill the form with the URL of the API endpoint (i.e. http://localhost:8000/api/hello if you're on localhost) and the access token coming from the part 1 of the tutorial. Going in the Django admin and get the token from there is not considered cheating, so it's an option.

Try performing a request and check that your *Resource Server* aka *Authorization Server* correctly responds with an HTTP 200.

Part 3 of the tutorial will show how to use an access token to authenticate users.

Part 3 - OAuth2 token authentication

Scenario

You want to use an *Access Token* to authenticate users against Django's authentication system.

Setup a provider

You need a fully-functional OAuth2 provider which is able to release access tokens: just follow the steps in the part 1 of the tutorial. To enable OAuth2 token authentication you need a middleware that checks for tokens inside requests and a custom authentication backend which takes care of token verification. In your settings.py:

```
AUTHENTICATION_BACKENDS = (
    'oauth2_provider.backends.OAuth2Backend',
    # Uncomment following if you want to access the admin
    #'django.contrib.auth.backends.ModelBackend'
    '...',
)

MIDDLEWARE_CLASSES = (
```

```
    '...',
    # If you use SessionAuthenticationMiddleware, be sure it appears before OAuth2TokenMiddleware.
    # SessionAuthenticationMiddleware is NOT required for using django-oauth-toolkit.
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'oauth2_provider.middleware.OAuth2TokenMiddleware',
    '...',
)
```

You will likely use the *django.contrib.auth.backends.ModelBackend* along with the OAuth2 backend (or you might not be able to log in into the admin), only pay attention to the order in which Django processes authentication backends.

If you put the OAuth2 backend *after* the AuthenticationMiddleware and *request.user* is valid, the backend will do nothing; if *request.user* is the Anonymous user it will try to authenticate the user using the OAuth2 access token.

If you put the OAuth2 backend *before* AuthenticationMiddleware, or AuthenticationMiddleware is not used at all, it will try to authenticate user with the OAuth2 access token and set *request.user* and *request._cached_user* fields so that AuthenticationMiddleware (when active) will not try to get user from the session.

If you use SessionAuthenticationMiddleware, be sure it appears before OAuth2TokenMiddleware. However SessionAuthenticationMiddleware is NOT required for using django-oauth-toolkit.

Protect your view

The authentication backend will run smoothly with, for example, *login_required* decorators, so that you can have a view like this in your *views.py* module:

```
from django.contrib.auth.decorators import login_required
from django.http.response import HttpResponse

@login_required()
def secret_page(request, *args, **kwargs):
    return HttpResponse('Secret contents!', status=200)
```

To check everything works properly, mount the view above to some url:

```
urlpatterns = [
    url(r'^secret$', 'my.views.secret_page', name='secret'),
    '...',
]
```

You should have an *Application* registered at this point, if you don't, follow the steps in the previous tutorials to create one. Obtain an *Access Token*, either following the OAuth2 flow of your application or manually creating in the Django admin. Now supposing your access token value is *123456* you can try to access your authenticated view:

```
curl -H "Authorization: Bearer 123456" -X GET http://localhost:8000/secret
```

Part 4 - Revoking an OAuth2 Token

Scenario

You've granted a user an *Access Token*, following part 1 and now you would like to revoke that token, probably in response to a client request (to logout).

Revoking a Token

Be sure that you've granted a valid token. If you've hooked in *oauth-toolkit* into your *urls.py* as specified in part 1, you'll have a URL at */o/revoke_token*. By submitting the appropriate request to that URL, you can revoke a user's *Access Token*.

Oauthlib is compliant with https://tools.ietf.org/html/rfc7009, so as specified, the revocation request requires:

- token: REQUIRED, this is the *Access Token* you want to revoke
- token_type_hint: OPTIONAL, designating either 'access_token' or 'refresh_token'.

Note that these revocation-specific parameters are in addition to the authentication parameters already specified by your particular client type.

Setup a Request

Depending on the client type you're using, the token revocation request you may submit to the authentication server may vary. A *Public* client, for example, will not have access to your *Client Secret*. A revoke request from a public client would omit that secret, and take the form:

```
POST /o/revoke_token/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
token=XXXX&client_id=XXXX
```

Where token is *Access Token* specified above, and client_id is the *Client id* obtained in obtained in part 1. If your application type is *Confidential* , it requires a *Client secret*, you will have to add it as one of the parameters:

```
POST /o/revoke_token/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
token=XXXX&client_id=XXXX&client_secret=XXXX
```

The server will respond wih a *200* status code on successful revocation. You can use *curl* to make a revoke request on your server. If you have access to a local installation of your authorization server, you can test revoking a token with a request like that shown below, for a *Confidential* client.

```
curl --data "token=XXXX&client_id=XXXX&client_secret=XXXX" http://localhost:8000/o/revoke_token/
```

Django Rest Framework

Getting started

Django OAuth Toolkit provide a support layer for Django REST Framework. This tutorial is based on the Django REST Framework example and shows you how to easily integrate with it.

NOTE

The following code has been tested with django 1.7.7 and Django REST Framework 3.1.1

Step 1: Minimal setup

Create a virtualenv and install following packages using *pip*...

```
pip install django-oauth-toolkit djangorestframework
```

Start a new Django project and add *'rest_framework'* and *'oauth2_provider'* to your *INSTALLED_APPS* setting.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    ...
    'oauth2_provider',
    'rest_framework',
)
```

Now we need to tell Django REST Framework to use the new authentication backend. To do so add the following lines at the end of your *settings.py* module:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'oauth2_provider.ext.rest_framework.OAuth2Authentication',
    )
}
```

Step 2: Create a simple API

Let's create a simple API for accessing users and groups.

Here's our project's root *urls.py* module:

```
from django.conf.urls import url, include
from django.contrib.auth.models import User, Group
from django.contrib import admin
admin.autodiscover()

from rest_framework import permissions, routers, serializers, viewsets

from oauth2_provider.ext.rest_framework import TokenHasReadWriteScope, TokenHasScope


# first we define the serializers
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User


class GroupSerializer(serializers.ModelSerializer):
    class Meta:
        model = Group


# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated, TokenHasReadWriteScope]
    queryset = User.objects.all()
    serializer_class = UserSerializer


class GroupViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated, TokenHasScope]
    required_scopes = ['groups']
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

```
# Routers provide an easy way of automatically determining the URL conf
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)
router.register(r'groups', GroupViewSet)


# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browseable API.
urlpatterns = [
    url(r'^', include(router.urls)),
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
    url(r'^admin/', include(admin.site.urls)),
]
```

Also add the following to your *settings.py* module:

```
OAUTH2_PROVIDER = {
    # this is the list of available scopes
    'SCOPES': {'read': 'Read scope', 'write': 'Write scope', 'groups': 'Access to your groups'}
}

REST_FRAMEWORK = {
    # ...

    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

OAUTH2_PROVIDER.SCOPES setting parameter contains the scopes that the application will be aware of, so we can use them for permission check.

Now run the following commands:

```
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

The first command creates the tables, the second creates the admin user account and the last one runs the application.

Next thing you should do is to login in the admin at

```
http://localhost:8000/admin
```

and create some users and groups that will be queried later through our API.

Step 3: Register an application

To obtain a valid access_token first we must register an application. DOT has a set of customizable views you can use to CRUD application instances, just point your browser at:

```
http://localhost:8000/o/applications/
```

Click on the link to create a new application and fill the form with the following data:

- Name: *just a name of your choice*
- Client Type: *confidential*
- Authorization Grant Type: *Resource owner password-based*

Save your app!

Step 4: Get your token and use your API

At this point we're ready to request an access_token. Open your shell

```
curl -X POST -d "grant_type=password&username=<user_name>&password=<password>" -u"<client_id>:<client
```

The *user_name* and *password* are the credential of the users registered in your *Authorization Server*, like any user created in Step 2. Response should be something like:

```
{
    "access_token": "<your_access_token>",
    "token_type": "Bearer",
    "expires_in": 36000,
    "refresh_token": "<your_refresh_token>",
    "scope": "read write groups"
}
```

Grab your access_token and start using your new OAuth2 API:

```
# Retrieve users
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/users/
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/users/1/

# Retrieve groups
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/groups/

# Insert a new user
curl -H "Authorization: Bearer <your_access_token>" -X POST -d"username=foo&password=bar" http://loca
```

Step 5: Testing Restricted Access

Let's try to access resources using a token with a restricted scope adding a *scope* parameter to the token request

```
curl -X POST -d "grant_type=password&username=<user_name>&password=<password>&scope=read" -u"<client_
```

As you can see the only scope provided is *read*:

```
{
    "access_token": "<your_access_token>",
    "token_type": "Bearer",
    "expires_in": 36000,
    "refresh_token": "<your_refresh_token>",
    "scope": "read"
}
```

We now try to access our resources:

```
# Retrieve users
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/users/
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/users/1/
```

Ok, this one works since users read only requires *read* scope.

```
# 'groups' scope needed
curl -H "Authorization: Bearer <your_access_token>" http://localhost:8000/groups/
```

```
# 'write' scope needed
curl -H "Authorization: Bearer <your_access_token>" -X POST -d"username=foo&password=bar" http://loca
```

You'll get a *"You do not have permission to perform this action"* error because your access_token does not provide the required scopes *groups* and *write*.

Permissions

Django OAuth Toolkit provides a few utility classes to use along with other permissions in Django REST Framework, so you can easily add scoped-based permission checks to your API views.

More details on how to add custom permissions to your API Endpoints can be found at the official Django REST Framework documentation

TokenHasScope

The *TokenHasScope* permission class allows the access only when the current access token has been authorized for **all** the scopes listed in the *required_scopes* field of the view.

For example:

```
class SongView(views.APIView):
    authentication_classes = [OAuth2Authentication]
    permission_classes = [TokenHasScope]
    required_scopes = ['music']
```

The *required_scopes* attribute is mandatory.

TokenHasReadWriteScope

The *TokenHasReadWriteScope* permission class allows the access based on the *READ_SCOPE* and *WRITE_SCOPE* configured in the settings.

When the current request's method is one of the "safe" methods *GET*, *HEAD*, *OPTIONS* the access is allowed only if the access token has been authorized for the *READ_SCOPE* scope. When the request's method is one of *POST*, *PUT*, *PATCH*, *DELETE* the access is allowed if the access token has been authorized for the *WRITE_SCOPE*.

The *required_scopes* attribute is optional and can be used to other scopes needed by the view.

For example:

```
class SongView(views.APIView):
    authentication_classes = [OAuth2Authentication]
    permission_classes = [TokenHasReadWriteScope]
    required_scopes = ['music']
```

When a request is performed both the *READ_SCOPE* \ *WRITE_SCOPE* and 'music' scopes are required to be authorized for the current access token.

TokenHasResourceScope

The *TokenHasResourceScope* permission class allows the access only when the current access token has been authorized for **all** the scopes listed in the *required_scopes* field of the view but according of request's method.

When the current request's method is one of the "safe" methods, the access is allowed only if the access token has been authorized for the *scope:read* scope (for example *music:read*). When the request's method is one of "non safe"

methods, the access is allowed only if the access token has been authorizes for the *scope:write* scope (for example *music:write*).

```
class SongView(views.APIView):
    authentication_classes = [OAuth2Authentication]
    permission_classes = [TokenHasResourceScope]
    required_scopes = ['music']
```

The *required_scopes* attribute is mandatory (you just need inform the resource scope).

IsAuthenticatedOrTokenHasScope

The *TokenHasResourceScope* permission class allows the access only when the current access token has been authorized for **all** the scopes listed in the *required_scopes* field of the view but according of request's method. And also allows access to Authenticated users who are authenticated in django, but were not authenticated trought the OAuth2Authentication class. This allows for protection of the api using scopes, but still let's users browse the full browseable API. To restrict users to only browse the parts of the browseable API they should be allowed to see, you can combine this wwith the DjangoModelPermission or the DjangoObjectPermission.

For example:

```
class SongView(views.APIView):
    permission_classes = [IsAuthenticatedOrTokenHasScope, DjangoModelPermission]
    required_scopes = ['music']
```

The *required_scopes* attribute is mandatory.

Using the views

Django OAuth Toolkit provides a set of pre-defined views for different purposes:

Function-based views

Django OAuth Toolkit provides decorators to help you in protecting your function-based views.

protected_resource(*scopes=None*, *validator_cls=OAuth2Validator*, *server_cls=Server*)

Decorator to protect views by providing OAuth2 authentication out of the box, optionally with scope handling. Basic usage, without using scopes:

```
from oauth2_provider.decorators import protected_resource

@protected_resource()
def my_view(request):
    # An access token is required to get here...
    # ...
    pass
```

If you want to check scopes as well when accessing a view you can pass them along as decorator's parameter:

```
from oauth2_provider.decorators import protected_resource

@protected_resource(scopes=['can_make_it can_break_it'])
def my_view(request):
    # An access token AND the right scopes are required to get here...
```

```
        # ...
        pass
```

The decorator also accept server and validator classes if you want or need to use your own OAuth2 logic:

```
from oauth2_provider.decorators import protected_resource
from myapp.oauth2_validators import MyValidator

@protected_resource(validator_cls=MyValidator)
def my_view(request):
    # You have to leverage your own logic to get here...
    # ...
    pass
```

rw_protected_resource(*scopes=None*, *validator_cls=OAuth2Validator*, *server_cls=Server*)
Decorator to protect views by providing OAuth2 authentication and read/write scopes out of the box. GET, HEAD, OPTIONS http methods require "read" scope. Otherwise "write" scope is required:

```
from oauth2_provider.decorators import rw_protected_resource

@rw_protected_resource()
def my_view(request):
    # If this is a POST, you have to provide 'write' scope to get here...
    # ...
    pass
```

If you need, you can ask for other scopes over "read" and "write":

```
from oauth2_provider.decorators import rw_protected_resource

@rw_protected_resource(scopes=['exotic_scope'])
def my_view(request):
    # If this is a POST, you have to provide 'exotic_scope write' scopes to get here...
    # ...
    pass
```

Class-based Views

Django OAuth Toolkit provides generic classes useful to implement OAuth2 protected endpoints using the *Class Based View* approach.

ProtectedResourceView(ProtectedResourceMixin, View):
A view that provides OAuth2 authentication out of the box. To implement a protected endpoint, just define your CBV as:

```
class MyEndpoint(ProtectedResourceView):
    """
    A GET endpoint that needs OAuth2 authentication
    """
    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

Please notice: `OPTION` method is not OAuth2 protected to allow preflight requests.

ScopedProtectedResourceView(ScopedResourceMixin, ProtectedResourceView):
A view that provides OAuth2 authentication and scopes handling out of the box. To implement a protected endpoint, just define your CBV specifying the `required_scopes` field:

```
class MyScopedEndpoint(ScopedProtectedResourceView):
    required_scopes = ['can_make_it can_break_it']

    """
    A GET endpoint that needs OAuth2 authentication
    and a set of scopes: 'can_make_it' and 'can_break_it'
    """
    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

ReadWriteScopedResourceView(ReadWriteScopedResourceMixin, ProtectedResourceView):
A view that provides OAuth2 authentication and read/write default scopes. `GET`, `HEAD`, `OPTIONS` http methods require `read` scope, others methods need the `write` scope. If you need, you can always specify an additional list of scopes in the `required_scopes` field:

```
class MyRWEndpoint(ReadWriteScopedResourceView):
    required_scopes = ['has_additional_powers']  # optional

    """
    A GET endpoint that needs OAuth2 authentication
    and the 'read' scope. If required_scopes was specified,
    clients also need those scopes.
    """
    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

Generic views in DOT are obtained composing a set of mixins you can find in the views.mixins module: feel free to use those mixins directly if you want to provide your own class based views.

Application Views

A set of views is provided to let users handle application instances without accessing Django Admin Site. Application views are listed at the url *applications/* and you can register a new one at the url *applications/register*. You can override default templates located in *templates/oauth2_provider* folder and provide a custom layout. Every view provides access only to data belonging to the logged in user who performs the request.

class `oauth2_provider.views.application.`**ApplicationDelete**(***kwargs*)
View used to delete an application owned by the request.user

class `oauth2_provider.views.application.`**ApplicationDetail**(***kwargs*)
Detail view for an application instance owned by the request.user

class `oauth2_provider.views.application.`**ApplicationList**(***kwargs*)
List view for all the applications owned by the request.user

class `oauth2_provider.views.application.`**ApplicationOwnerIsUserMixin**
This mixin is used to provide an Application queryset filtered by the current request.user.

class `oauth2_provider.views.application.`**ApplicationRegistration**(***kwargs*)
View used to register a new Application for the request.user

get_form_class()
Returns the form class for the application model

class `oauth2_provider.views.application.`**ApplicationUpdate**(***kwargs*)
View used to update an application owned by the request.user

Granted Tokens Views

A set of views is provided to let users handle tokens that have been granted to them, without needing to accessing Django Admin Site. Every view provides access only to the tokens that have been granted to the user performing the request.

Granted Token views are listed at the url *authorized_tokens/*.

For each granted token there is a delete view that allows you to delete such token. You can override default templates *authorized-tokens.html* for the list view and *authorized-token-delete.html* for the delete view; they are located inside *templates/oauth2_provider* folder.

class `oauth2_provider.views.token.`**`AuthorizedTokenDeleteView`**(***kwargs*)
View for revoking a specific token

`model`
alias of `AccessToken`

class `oauth2_provider.views.token.`**`AuthorizedTokensListView`**(***kwargs*)
Show a page where the current logged-in user can see his tokens so they can revoke them

`get_queryset`()
Show only user's tokens

`model`
alias of `AccessToken`

Mixins for Class Based Views

class `oauth2_provider.views.mixins.`**`OAuthLibMixin`**
This mixin decouples Django OAuth Toolkit from OAuthLib.

Users can configure the Server, Validator and OAuthlibCore classes used by this mixin by setting the following class variables:

- server_class
- validator_class
- oauthlib_backend_class

`create_authorization_response`(*request*, *scopes*, *credentials*, *allow*)
A wrapper method that calls create_authorization_response on *server_class* instance.

Parameters

- **`request`** – The current django.http.HttpRequest object
- **`scopes`** – A space-separated string of provided scopes
- **`credentials`** – Authorization credentials dictionary containing *client_id*, *state*, *redirect_uri*, *response_type*
- **`allow`** – True if the user authorize the client, otherwise False

`create_revocation_response`(*request*)
A wrapper method that calls create_revocation_response on the *server_class* instance.

Parameters **`request`** – The current django.http.HttpRequest object

`create_token_response`(*request*)
A wrapper method that calls create_token_response on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

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

Parameters **error** – OAuthToolkitError

classmethod get_oauthlib_backend_class()
Return the OAuthLibCore implementation class to use

classmethod get_oauthlib_core()
Cache and return *OAuthlibCore* instance so it will be created only on first request

get_scopes()
This should return the list of scopes required to access the resources. By default it returns an empty list

classmethod get_server()
Return an instance of *server_class* initialized with a *validator_class* object

classmethod get_server_class()
Return the OAuthlib server class to use

classmethod get_validator_class()
Return the RequestValidator implementation class to use

validate_authorization_request(*request*)
A wrapper method that calls validate_authorization_request on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

verify_request(*request*)
A wrapper method that calls verify_request on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

class oauth2_provider.views.mixins.**ProtectedResourceMixin**
Helper mixin that implements OAuth2 protection on request dispatch, specially useful for Django Generic Views

class oauth2_provider.views.mixins.**ReadWriteScopedResourceMixin**
Helper mixin that implements "read and write scopes" behavior

class oauth2_provider.views.mixins.**ScopedResourceMixin**
Helper mixin that implements "scopes handling" behaviour

get_scopes(**args*, ***kwargs*)
Return the scopes needed to access the resource

Parameters **args** – Support scopes injections from the outside (not yet implemented)

Views code and details

Generic

Generic views are intended to use in a "batteries included" fashion to protect own views with OAuth2 authentication and Scopes handling.

class oauth2_provider.views.generic.**ProtectedResourceView**(***kwargs*)
Generic view protecting resources by providing OAuth2 authentication out of the box

oauthlib_backend_class
alias of OAuthLibCore

server_class
alias of Server

class oauth2_provider.views.generic.**ReadWriteScopedResourceView**(***kwargs*)
Generic view protecting resources with OAuth2 authentication and read/write scopes. GET, HEAD, OPTIONS http methods require "read" scope. Otherwise "write" scope is required.

class oauth2_provider.views.generic.**ScopedProtectedResourceView**(***kwargs*)
Generic view protecting resources by providing OAuth2 authentication and Scopes handling out of the box

Mixins

These views are mainly for internal use, but advanced users may use them as basic components to customize OAuth2 logic inside their Django applications.

class oauth2_provider.views.mixins.**OAuthLibMixin**
This mixin decouples Django OAuth Toolkit from OAuthLib.

Users can configure the Server, Validator and OAuthlibCore classes used by this mixin by setting the following class variables:

•server_class

•validator_class

•oauthlib_backend_class

create_authorization_response(*request*, *scopes*, *credentials*, *allow*)
A wrapper method that calls create_authorization_response on *server_class* instance.

Parameters

- **request** – The current django.http.HttpRequest object
- **scopes** – A space-separated string of provided scopes
- **credentials** – Authorization credentials dictionary containing *client_id*, *state*, *redirect_uri*, *response_type*
- **allow** – True if the user authorize the client, otherwise False

create_revocation_response(*request*)
A wrapper method that calls create_revocation_response on the *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

create_token_response(*request*)
A wrapper method that calls create_token_response on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

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

Parameters **error** – OAuthToolkitError

classmethod get_oauthlib_backend_class()
Return the OAuthLibCore implementation class to use

classmethod get_oauthlib_core()
Cache and return *OAuthlibCore* instance so it will be created only on first request

get_scopes()
This should return the list of scopes required to access the resources. By default it returns an empty list

classmethod get_server()
Return an instance of *server_class* initialized with a *validator_class* object

classmethod get_server_class()
Return the OAuthlib server class to use

classmethod get_validator_class()
Return the RequestValidator implementation class to use

validate_authorization_request(*request*)
A wrapper method that calls validate_authorization_request on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

verify_request(*request*)
A wrapper method that calls verify_request on *server_class* instance.

Parameters **request** – The current django.http.HttpRequest object

class oauth2_provider.views.mixins.**ProtectedResourceMixin**
Helper mixin that implements OAuth2 protection on request dispatch, specially useful for Django Generic Views

class oauth2_provider.views.mixins.**ReadWriteScopedResourceMixin**
Helper mixin that implements "read and write scopes" behavior

class oauth2_provider.views.mixins.**ScopedResourceMixin**
Helper mixin that implements "scopes handling" behaviour

get_scopes(**args*, ***kwargs*)
Return the scopes needed to access the resource

Parameters **args** – Support scopes injections from the outside (not yet implemented)

Base

Views needed to implement the main OAuth2 authorization flows supported by Django OAuth Toolkit.

class oauth2_provider.views.base.**AuthorizationView**(***kwargs*)
Implements and endpoint to handle *Authorization Requests* as in RFC6749 Section 4.1.1 and prompting the user with a form to determine if she authorizes the client application to access her data. This endpoint is reached two times during the authorization process: * first receive a `GET` request from user asking authorization for a certain client application, a form is served possibly showing some useful info and prompting for *authorize/do not authorize*.

•then receive a `POST` request possibly after user authorized the access

Some informations contained in the `GET` request and needed to create a Grant token during the `POST` request would be lost between the two steps above, so they are temporary stored in hidden fields on the form. A possible alternative could be keeping such informations in the session.

The endpoint is used in the followin flows: * Authorization code * Implicit grant

oauthlib_backend_class
alias of `OAuthLibCore`

server_class
alias of `Server`

class oauth2_provider.views.base.**BaseAuthorizationView**(***kwargs*)
Implements a generic endpoint to handle *Authorization Requests* as in RFC6749 Section 4.1.1. The view does not implement any strategy to determine *authorize/do not authorize* logic. The endpoint is used in the following flows:

- Authorization code
- Implicit grant

error_response(*error*, ***kwargs*)
Handle errors either by redirecting to redirect_uri with a json in the body containing error details or providing an error response

class oauth2_provider.views.base.**RevokeTokenView**(***kwargs*)
Implements an endpoint to revoke access or refresh tokens

oauthlib_backend_class
alias of OAuthLibCore

server_class
alias of Server

class oauth2_provider.views.base.**TokenView**(***kwargs*)
Implements an endpoint to provide access tokens

The endpoint is used in the following flows: * Authorization code * Password * Client credentials

oauthlib_backend_class
alias of OAuthLibCore

server_class
alias of Server

Models

class oauth2_provider.models.**AbstractApplication**(**args*, ***kwargs*)
An Application instance represents a Client on the Authorization server. Usually an Application is created manually by client's developers after logging in on an Authorization Server.

Fields:

- **client_id The client identifier issued to the client during the** registration process as described in RFC6749 Section 2.2
- user ref to a Django user
- **redirect_uris The list of allowed redirect uri. The string** consists of valid URLs separated by space
- client_type Client type as described in RFC6749 Section 2.1
- **authorization_grant_type Authorization flows available to the** Application
- **client_secret Confidential secret issued to the client during** the registration process as described in RFC6749 Section 2.2
- name Friendly name for the Application

default_redirect_uri
Returns the default redirect_uri extracting the first item from the redirect_uris string

redirect_uri_allowed(*uri*)
Checks if given url is one of the items in redirect_uris string

Parameters **uri** – Url to check

class oauth2_provider.models.**AccessToken**(**args*, ***kwargs*)

An AccessToken instance represents the actual access token to access user's resources, as in RFC6749 Section 5.

Fields:

- `user` The Django user representing resources' owner
- `token` Access token
- `application` Application instance
- `expires` Date and time of token expiration, in DateTime format
- `scope` Allowed scopes

allow_scopes(*scopes*)

Check if the token allows the provided scopes

Parameters **scopes** – An iterable containing the scopes to check

is_expired()

Check token expiration with timezone awareness

is_valid(*scopes=None*)

Checks if the access token is valid.

Parameters **scopes** – An iterable containing the scopes to check or None

revoke()

Convenience method to uniform tokens' interface, for now simply remove this token from the database in order to revoke it.

scopes

Returns a dictionary of allowed scope names (as keys) with their descriptions (as values)

class oauth2_provider.models.**Application**(*id*, *client_id*, *user*, *redirect_uris*, *client_type*, *authorization_grant_type*, *client_secret*, *name*, *skip_authorization*)

class oauth2_provider.models.**Grant**(**args*, ***kwargs*)

A Grant instance represents a token with a short lifetime that can be swapped for an access token, as described in RFC6749 Section 4.1.2

Fields:

- `user` The Django user who requested the grant
- `code` The authorization code generated by the authorization server
- `application` Application instance this grant was asked for
- **`expires` Expire time in seconds, defaults to** `settings.AUTHORIZATION_CODE_EXPIRE_SECONDS`
- `redirect_uri` Self explained
- `scope` Required scopes, optional

is_expired()

Check token expiration with timezone awareness

class oauth2_provider.models.**RefreshToken**(**args*, ***kwargs*)

A RefreshToken instance represents a token that can be swapped for a new access token when it expires.

Fields:

- `user` The Django user representing resources' owner

- `token` Token value
- `application` Application instance
- **`access_token` AccessToken instance this refresh token is** bounded to

`revoke`()
Delete this refresh token along with related access token

`oauth2_provider.models.`**`get_application_model`**()
Return the Application model that is active in this project.

Advanced topics

Extending the Application model

An Application instance represents a *Client* on the *Authorization server*. Usually an Application is issued to client's developers after they log in on an Authorization Server and pass in some data which identify the Application itself (let's say, the application name). Django OAuth Toolkit provides a very basic implementation of the Application model containing only the data strictly required during all the OAuth processes but you will likely need some extra info, like application logo, acceptance of some user agreement and so on.

class `AbstractApplication`(*models.Model*)
This is the base class implementing the bare minimum for Django OAuth Toolkit to work

- `client_id` The client identifier issued to the client during the registration process as described in RFC6749 Section 2.2
- `user` ref to a Django user
- `redirect_uris` The list of allowed redirect uri. The string consists of valid URLs separated by space
- `client_type` Client type as described in RFC6749 Section 2.1
- `authorization_grant_type` Authorization flows available to the Application
- `client_secret` Confidential secret issued to the client during the registration process as described in RFC6749 Section 2.2
- `name` Friendly name for the Application

Django OAuth Toolkit lets you extend the AbstractApplication model in a fashion like Django's custom user models.

If you need, let's say, application logo and user agreement acceptance field, you can to this in your Django app (provided that your app is in the list of the INSTALLED_APPS in your settings module):

```
from django.db import models
from oauth2_provider.models import AbstractApplication

class MyApplication(AbstractApplication):
    logo = models.ImageField()
    agree = models.BooleanField()
```

Then you need to tell Django OAuth Toolkit which model you want to use to represent applications. Write something like this in your settings module:

```
OAUTH2_PROVIDER_APPLICATION_MODEL='your_app_name.MyApplication'
```

Be aware that, when you intend to swap the application model, you should create and run the migration defining the swapped application model prior to setting OAUTH2_PROVIDER_APPLICATION_MODEL. You'll run into models.E022 in Core system checks if you don't get the order right.

That's all, now Django OAuth Toolkit will use your model wherever an Application instance is needed.

> **Notice:** *OAUTH2_PROVIDER_APPLICATION_MODEL* is the only setting variable that is not namespaced, this is because of the way Django currently implements swappable models. See issue #90 (https://github.com/evonove/django-oauth-toolkit/issues/90) for details

Skip authorization form

Depending on the OAuth2 flow in use and the access token policy, users might be prompted for the same authorization multiple times: sometimes this is acceptable or even desirable but other times it isn't. To control DOT behaviour you can use the *approval_prompt* parameter when hitting the authorization endpoint. Possible values are:

- *force* - users are always prompted for authorization.
- *auto* - users are prompted only the first time, subsequent authorizations for the same application and scopes will be automatically accepted.

Skip authorization completely for trusted applications

You might want to completely bypass the authorization form, for instance if your application is an in-house product or if you already trust the application owner by other means. To this end, you have to set `skip_authorization = True` on the `Application` model, either programmaticaly or within the Django admin. Users will *not* be prompted for authorization, even on the first use of the application.

Settings

Our configurations are all namespaced under the *OAUTH2_PROVIDER* settings with the solely exception of *OAUTH2_PROVIDER_APPLICATION_MODEL*: this is because of the way Django currently implements swappable models. See issue #90 (https://github.com/evonove/django-oauth-toolkit/issues/90) for details.

For example:

```
OAUTH2_PROVIDER = {
    'SCOPES': {
        'read': 'Read scope',
        'write': 'Write scope',
    },

    'CLIENT_ID_GENERATOR_CLASS': 'oauth2_provider.generators.ClientIdGenerator',

}
```

A big *thank you* to the guys from Django REST Framework for inspiring this.

List of available settings

ACCESS_TOKEN_EXPIRE_SECONDS

The number of seconds an access token remains valid. Requesting a protected resource after this duration will fail. Keep this value high enough so clients can cache the token for a reasonable amount of time.

APPLICATION_MODEL

The import string of the class (model) representing your applications. Overwrite this value if you wrote your own implementation (subclass of `oauth2_provider.models.Application`).

AUTHORIZATION_CODE_EXPIRE_SECONDS

The number of seconds an authorization code remains valid. Requesting an access token after this duration will fail. RFC6749 Section 4.1.2 recommends a 10 minutes (600 seconds) duration.

CLIENT_ID_GENERATOR_CLASS

The import string of the class responsible for generating client identifiers. These are usually random strings.

CLIENT_SECRET_GENERATOR_CLASS

The import string of the class responsible for generating client secrets. These are usually random strings.

CLIENT_SECRET_GENERATOR_LENGTH

The length of the generated secrets, in characters. If this value is too low, secrets may become subject to bruteforce guessing.

OAUTH2_SERVER_CLASS

The import string for the `server_class` (or `oauthlib.oauth2.Server` subclass) used in the `OAuthLibMixin` that implements OAuth2 grant types.

OAUTH2_VALIDATOR_CLASS

The import string of the `oauthlib.oauth2.RequestValidator` subclass that validates every step of the OAuth2 process.

OAUTH2_BACKEND_CLASS

The import string for the `oauthlib_backend_class` used in the `OAuthLibMixin`, to get a `Server` instance.

SCOPES

A dictionary mapping each scope name to its human description.

DEFAULT_SCOPES

A list of scopes that should be returned by default. This is a subset of the keys of the SCOPES setting. By default this is set to '__all__' meaning that the whole set of SCOPES will be returned.

```
DEFAULT_SCOPES = ['read', 'write']
```

READ_SCOPE

The name of the *read* scope.

WRITE_SCOPE

The name of the *write* scope.

REFRESH_TOKEN_EXPIRE_SECONDS

The number of seconds before a refresh token gets removed from the database by the `cleartokens` management command. Check *cleartokens* management command for further info.

ROTATE_REFRESH_TOKEN

When is set to *True* (default) a new refresh token is issued to the client when the client refreshes an access token.

REQUEST_APPROVAL_PROMPT

Can be `'force'` or `'auto'`. The strategy used to display the authorization form. Refer to *Skip authorization form*.

Management commands

Django OAuth Toolkit exposes some useful management commands that can be run via shell or by other means (eg: cron)

cleartokens

The `cleartokens` management command allows the user to remove those refresh tokens whose lifetime is greater than the amount specified by `REFRESH_TOKEN_EXPIRE_SECONDS` settings. It is important that this command is run regularly (eg: via cron) to avoid cluttering the database with expired refresh tokens.

If `cleartokens` runs daily the maximum delay before a refresh token is removed is `REFRESH_TOKEN_EXPIRE_SECONDS` + 1 day. This is normally not a problem since refresh tokens are long lived.

Note: Refresh tokens need to expire before AccessTokens can be removed from the database. Using `cleartokens` without `REFRESH_TOKEN_EXPIRE_SECONDS` has limited effect.

Glossary

Authorization Server The authorization server asks resource owners for their consensus to let client applications access their data. It also manages and issues the tokens needed for all the authorization flows supported by OAuth2 spec. Usually the same application offering resources through an OAuth2-protected API also behaves like an authorization server.

Resource Server An application providing access to its own resources through an API protected following the OAuth2 spec.

Application An Application represents a Client on the Authorization server. Usually an Application is created manually by client's developers after logging in on an Authorization Server.

Client A client is an application authorized to access OAuth2-protected resources on behalf and with the authorization of the resource owner.

Resource Owner The user of an application which exposes resources to third party applications through OAuth2. The resource owner must give her authorization for third party applications to be able to access her data.

Access Token A token needed to access resources protected by OAuth2. It has a lifetime which is usually quite short.

Authorization Code The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. It is used to authenticate the client and grant the transmission of the Access Token.

Authorization Token A token the authorization server issues to clients that can be swapped for an access token. It has a very short lifetime since the swap has to be performed shortly after users provide their authorization.

Refresh Token A token the authorization server may issue to clients and can be swapped for a brand new access token, without repeating the authorization process. It has no expire time.

Contributing

Setup

Fork *django-oauth-toolkit* repository on GitHub and follow these steps:

- Create a virtualenv and activate it
- Clone your repository locally
- cd into the repository and type *pip install -r requirements/optional.txt* (this will install both optional and base requirements, useful during development)

Issues

You can find the list of bugs, enhancements and feature requests on the issue tracker. If you want to fix an issue, pick up one and add a comment stating you're working on it. If the resolution implies a discussion or if you realize the comments on the issue are growing pretty fast, move the discussion to the Google Group.

Pull requests

Please avoid providing a pull request from your *master* and use **topic branches** instead; you can add as many commits as you want but please keep them in one branch which aims to solve one single issue. Then submit your pull request. To create a topic branch, simply do:

```
git checkout -b fix-that-issue
Switched to a new branch 'fix-that-issue'
```

When you're ready to submit your pull request, first push the topic branch to your GitHub repo:

```
git push origin fix-that-issue
```

Now you can go to your repository dashboard on GitHub and open a pull request starting from your topic branch. You can apply your pull request to the *master* branch of django-oauth-toolkit (this should be the default behaviour of GitHub user interface).

Next you should add a comment about your branch, and if the pull request refers to a certain issue, insert a link to it. The repo managers will be notified of your pull request and it will be reviewed, in the meantime 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 regularly

It's a good practice to pull upstream changes from master into your fork on a regular basis, in fact if you work on outdated code and your changes diverge too far from master, the pull request has to be rejected.

To pull in upstream changes:

```
git remote add upstream https://github.com/evonove/django-oauth-toolkit.git
git fetch upstream
```

Then merge the changes that you fetched:

```
git merge upstream/master
```

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

Note: Please be sure to rebase your commits on the master when possible, so your commits can be fast-forwarded: we try to avoid *merge commits* when they are not necessary.

How to get your pull request accepted

We really want your code, so please follow these simple guidelines to make the process as smooth as possible.

Run the tests!

Django OAuth Toolkit aims to support different Python and Django versions, so we use **tox** to run tests on multiple configurations. At any time during the development and at least before submitting the pull request, please run the testsuite via:

```
tox
```

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

Add the tests!

Whenever you add code, you have to add tests as well. We cannot accept untested code, so unless it is a peculiar situation you previously discussed with the core committers, if your pull request reduces the test coverage it will be **immediately rejected**.

Code conventions matter

There are no good nor bad conventions, just follow PEP8 (run some lint tool for this) and nobody will argue. Try reading our code and grasp the overall philosophy regarding method and variable names, avoid *black magics* for the

sake of readability, keep in mind that *simple is better than complex*. If you feel the code is not straightforward, add a comment. If you think a function is not trivial, add a docstrings.

The contents of this page are heavily based on the docs from django-admin2

Changelog

0.11.0 [2016-12-1]

- #424: Added a ROTATE_REFRESH_TOKEN setting to control whether refresh tokens are reused or not
- #315: AuthorizationView does not overwrite requests on get
- #425: Added support for Django 1.10
- #396: Added an IsAuthenticatedOrTokenHasScope Permission
- #357: Support multiple-user clients by allowing User to be NULL for Applications
- #389: Reuse refresh tokens if enabled.

0.10.0 [2015-12-14]

- **#322: dropping support for python 2.6 and django 1.4, 1.5, 1.6**
- #310: Fixed error that could occur sometimes when checking validity of incomplete AccessToken/Grant
- #333: Added possibility to specify the default list of scopes returned when scope parameter is missing
- #325: Added management views of issued tokens
- #249: Added a command to clean expired tokens
- #323: Application registration view uses custom application model in form class
- #299: 'server_class' is now pluggable through Django settings
- #309: Add the py35-django19 env to travis
- #308: Use compact syntax for tox envs
- #306: Django 1.9 compatibility
- #288: Put additional information when generating token responses
- #297: Fixed doc about SessionAuthenticationMiddleware
- #273: Generic read write scope by resource

0.9.0 [2015-07-28]

- `oauthlib_backend_class` is now pluggable through Django settings
- #127: `application/json` Content-Type is now supported using `JSONOAuthLibCore`
- #238: Fixed redirect uri handling in case of error
- #229: Invalidate access tokens when getting a new refresh token
- added support for oauthlib 1.0

0.8.2 [2015-06-25]

- Fix the migrations to be two-step and allow upgrade from 0.7.2

0.8.1 [2015-04-27]

- South migrations fixed. Added new django migrations.

0.8.0 [2015-03-27]

- Several docs improvements and minor fixes
- #185: fixed vulnerabilities on Basic authentication
- #173: ProtectResourceMixin now allows OPTIONS requests
- Fixed client_id and client_secret characters set
- #169: hide sensitive informations in error emails
- #161: extend search to all token types when revoking a token
- #160: return empty response on successful token revocation
- #157: skip authorization form with `skip_authorization_completely` class field
- #155: allow custom uri schemes
- fixed `get_application_model` on Django 1.7
- fixed non rotating refresh tokens
- #137: fixed base template
- customized `client_secret` lenght
- #38: create access tokens not bound to a user instance for *client credentials* flow

0.7.2 [2014-07-02]

- Don't pin oauthlib

0.7.0 [2014-03-01]

- Created a setting for the default value for approval prompt.
- Improved docs
- Don't pin django-braces and six versions

Backwards incompatible changes in 0.7.0

- Make Application model truly "swappable" (introduces a new non-namespaced setting OAUTH2_PROVIDER_APPLICATION_MODEL)

0.6.1 [2014-02-05]

- added support for *scope* query parameter keeping backwards compatibility for the original *scopes* parameter.
- __str__ method in Application model returns name when available

0.6.0 [2014-01-26]

- oauthlib 0.6.1 support
- Django dev branch support
- Python 2.6 support
- Skip authorization form via *approval_prompt* parameter

Bugfixes

- Several fixes to the docs
- Issue #71: Fix migrations
- Issue #65: Use OAuth2 password grant with multiple devices
- Issue #84: Add information about login template to tutorial.
- Issue #64: Fix urlencode clientid secret

0.5.0 [2013-09-17]

- oauthlib 0.6.0 support

Backwards incompatible changes in 0.5.0

- backends.py module has been renamed to oauth2_backends.py so you should change your imports whether you're extending this module

Bugfixes

- Issue #54: Auth backend proposal to address #50
- Issue #61: Fix contributing page
- Issue #55: Add support for authenticating confidential client with request body params
- Issue #53: Quote characters in the url query that are safe for Django but not for oauthlib

0.4.1 [2013-09-06]

- Optimize queries on access token validation

0.4.0 [2013-08-09]

New Features

- Add Application management views, you no more need the admin to register, update and delete your application.
- Add support to configurable application model
- Add support for function based views

Backwards incompatible changes in 0.4.0

- *SCOPE* attribute in settings is now a dictionary to store *{'scope_name': 'scope_description'}*
- Namespace 'oauth2_provider' is mandatory in urls. See issue #36

Bugfixes

- Issue #25: Bug in the Basic Auth parsing in Oauth2RequestValidator
- Issue #24: Avoid generation of client_id with ":" colon char when using HTTP Basic Auth
- Issue #21: IndexError when trying to authorize an application
- Issue #9: Default_redirect_uri is mandatory when grant_type is implicit, authorization_code or all-in-one
- Issue #22: Scopes need a verbose description
- Issue #33: Add django-oauth-toolkit version on example main page
- Issue #36: Add mandatory namespace to urls
- Issue #31: Add docstring to OAuthToolkitError and FatalClientError
- Issue #32: Add docstring to validate_uris
- Issue #34: Documentation tutorial part1 needs corsheaders explanation
- Issue #36: Add mandatory namespace to urls
- Issue #45: Add docs for AbstractApplication
- Issue #47: Add docs for views decorators

0.3.2 [2013-07-10]

- Bugfix #37: Error in migrations with custom user on Django 1.5

0.3.1 [2013-07-10]

- Bugfix #27: OAuthlib refresh token refactoring

0.3.0 [2013-06-14]

- Django REST Framework integration layer
- Bugfix #13: Populate request with client and user in validate_bearer_token
- Bugfix #12: Fix paths in documentation

Backwards incompatible changes in 0.3.0

- *requested_scopes* parameter in ScopedResourceMixin changed to *required_scopes*

0.2.1 [2013-06-06]

- Core optimizations

0.2.0 [2013-06-05]

- Add support for Django1.4 and Django1.6
- Add support for Python 3.3
- Add a default ReadWriteScoped view
- Add tutorial to docs

0.1.0 [2013-05-31]

- Support OAuth2 Authorization Flows

0.0.0 [2013-05-17]

- Discussion with Daniel Greenfeld at Django Circus
- Ignition

CHAPTER 4

Indices and tables

- genindex
- modindex

O

A

B

C

D

E

G