drf-social-oauth2

Release 3.1.0

Wagner de Lima

Apr 26, 2026

CONTENTS

DRF-Social-OAuth2 is a powerful module that enables OAuth2 social authentication for applications built on Django REST Framework. By providing seamless integration with python-social-auth and django-oauth-toolkit, this package facilitates the setup of social authentication for your REST API, as well as your OAuth2 provider.

If you're new to OAuth2 or find it challenging to understand, don't worry. DRF-Social-OAuth2 offers a straightforward and streamlined approach to OAuth2 authentication. However, we highly recommend familiarizing yourself with the OAuth2 concepts and terminology by referring to our recommended resources or other online tutorials.

If you're eager to test out DRF-Social-OAuth2 but do not want to go through the trouble of setting it up in your local environment, you can visit our facebook setup repository. It contains all the necessary configurations for you to get started. You'll only need to add a database configuration to your settings.py file, and you'll be ready to go! Generally speaking, it will help you set up your own django + drf-social-oauth2 project.

We hope that DRF-Social-OAuth2 simplifies your social authentication process and enhances the security and usability of your Django REST Framework application. If you encounter any issues or have suggestions, feel free to submit them to our GitHub repository or reach out to us via our support channels.

CHAPTER

ONE

CONTENTS

1.1 Usage

This guide will walk you through the process of installing DRF-Social-OAuth2 and setting it up for use with your Django REST Framework application. It assumes that you have some familiarity with Django and have a basic understanding of OAuth2 authentication. If you're new to Django or OAuth2, we recommend checking out our resources section for additional learning materials.

To begin, you'll need to have Python 3 and pip installed on your local machine. Once you have those installed, you can follow the instructions outlined in the installation section of this guide to install DRF-Social-OAuth2 and its dependencies. Then, you can configure your Django settings to use DRF-Social-OAuth2 by adding the necessary lines to your settings.py file. Finally, you'll need to migrate your database to apply the changes.

By the end of this guide, you'll have successfully installed DRF-Social-OAuth2 and set it up for use with your Django REST Framework application. With DRF-Social-OAuth2, you can make your REST API more secure and user-friendly by allowing users to authenticate with their social media accounts.

1.1.1 Installation

This framework is published at the PyPI, install it with pip:

```
$ pip install drf_social_oauth2==3.1.0
```

To enable OAuth2 social authentication support for your Django REST Framework application, you need to install and configure drf-social-oauth2. To get started, add the following packages to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'oauth2_provider',
    'social_django',
    'drf_social_oauth2',
)
```

Include social auth urls to your urls.py:

```
from django.conf.urls import url

urlpatterns = patterns(
    ...
    url(r'^auth/', include('drf_social_oauth2.urls', namespace='drf'))
)
```

For versions of Django 4.0 or higher, use *re_path* instead:

```
from django.urls import re_path

urlpatterns = patterns(
    ...
    re_path(r'^auth/', include('drf_social_oauth2.urls', namespace='drf'))
)
```

Next, add the following context processors to your TEMPLATE_CONTEXT_PROCESSORS:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social_django.context_processors.backends',
    'social_django.context_processors.login_redirect',
)
```

Note that since Django version 1.8, the TEMPLATE_CONTEXT_PROCESSORS setting is deprecated. You should instead set the 'context_processors' option in the OPTIONS of a DjangoTemplates backend:

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
        },
    }
]
```

You can then enable the authentication classes for Django REST Framework by default or per view by updating the REST_FRAMEWORK and AUTHENTICATION_BACKENDS entries in your settings.py:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        # 'oauth2_provider.ext.rest_framework.OAuth2Authentication',  # django-oauth-
↪toolkit < 1.0.0
        'oauth2_provider.contrib.rest_framework.OAuth2Authentication',  # django-oauth-
↪toolkit >= 1.0.0
        'drf_social_oauth2.authentication.SocialAuthentication',
    ),
}
```

```
AUTHENTICATION_BACKENDS = (
    ...
   'drf_social_oauth2.backends.DjangoOAuth2',
   'django.contrib.auth.backends.ModelBackend',
)
```

The following are settings available for drf-social-oauth2:

- DRFSO2_PROPRIETARY_BACKEND_NAME: name of your OAuth2 social backend (e.g "Facebook"), defaults to "Django"
- DRFSO2_URL_NAMESPACE: namespace for reversing URLs
- ACTIVATE_JWT: if set to True, both access and refresh tokens are issued as JWTs signed with SECRET_KEY (HS256). Default is False. See *Activating JWT tokens* below.

1.1.2 Activating JWT tokens

To make /auth/token and /auth/convert-token issue JWT-encoded access and refresh tokens, set ACTIVATE_JWT = True in your Django settings:

```
# settings.py
ACTIVATE_JWT = True
```

That is the only thing you need to do — provided 'drf_social_oauth2' is in INSTALLED_APPS, the package's AppConfig.ready() hook wires the JWT generators into django-oauth-toolkit at startup.

What the response looks like

The token_type field stays "Bearer" — that is the OAuth2 transport, not the token format. The JWT lives in the access_token (and refresh_token) value itself: three dot-separated base64url segments.

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbiI6Ii4uLiJ9.SIGNATURE
↪",
    "expires_in": 3600,
    "token_type": "Bearer",
    "scope": "read write",
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbiI6Ii4uLiJ9.SIGNATURE
↪"
}
```

You can verify the token format with PyJWT:

```
import jwt
from django.conf import settings

decoded = jwt.decode(access_token, settings.SECRET_KEY, algorithms=['HS256'])
# {'token': '<random 30-char opaque token>'}
```

Using the token in subsequent requests

Send it as a normal Bearer credential:

```
Authorization: Bearer eyJhbGciOi...
```

Do **not** include the backend name in the header (e.g. Bearer facebook eyJ...); that format is rejected by OAuth2Authentication.

Caveat: explicit overrides win

If your OAUTH2_PROVIDER dict explicitly sets ACCESS_TOKEN_GENERATOR or REFRESH_TOKEN_GENERATOR, those values take precedence and ACTIVATE_JWT becomes a no-op. Either remove those keys or point them at 'drf_social_oauth2.generate_token' yourself.

1.2 Setting Up a New Application

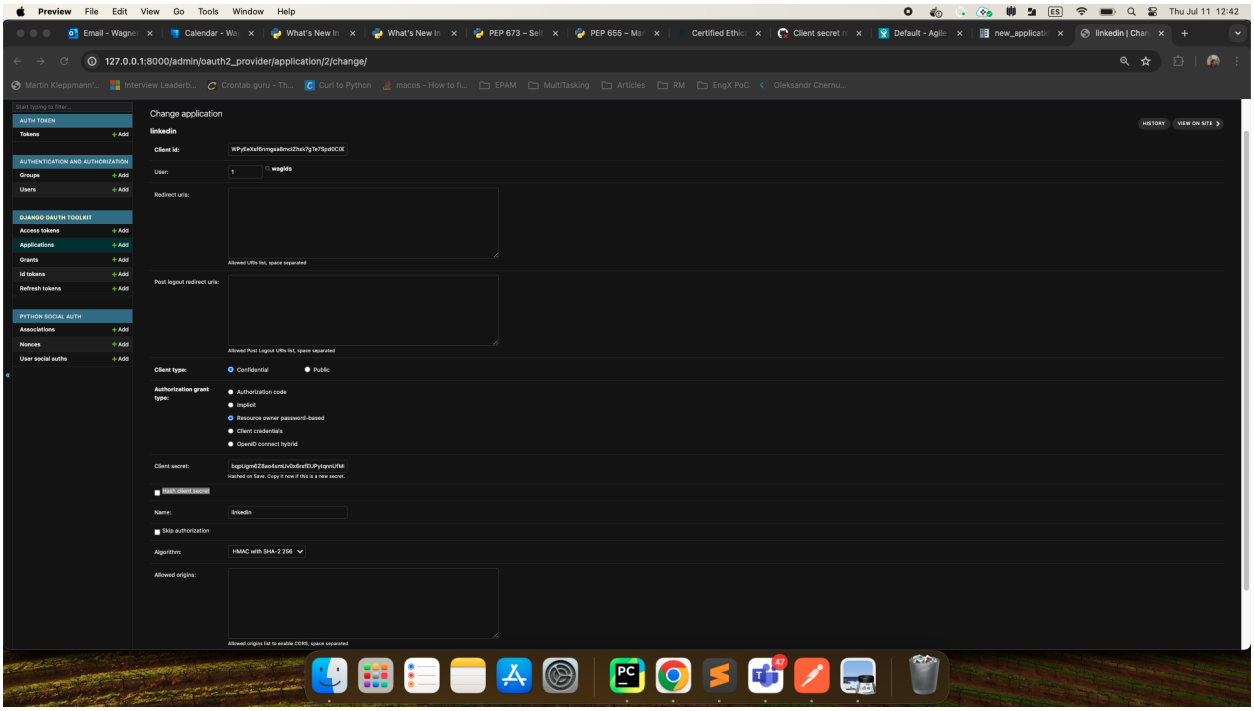
To begin, navigate to the Django admin panel and create a new application with the following configuration:

- Leave the *client_id* and *client_secret* fields unchanged.
- Set the *user* field to your superuser.
- Leave the *redirect_uris* field blank.
- Set the *client_type* field to confidential.
- Set the *authorization_grant_type* field to 'Resource owner password-based'.
- Optionally, you can set the *name* field to a name of your choice.

With these settings in place, the installation is now complete and you can proceed to test the newly configured application.

For further information and to take full advantage of the capabilities of this package, it is highly recommended that you refer to the documentation for python-social-auth and django-oauth-toolkit. If you intend to enable a social backend such as Facebook, you may want to consult the python-social-auth documentation on supported backends and the django-social-auth documentation on backend configuration.

Screenshot of the new application creation.



In your new application, you will see a Hash client secret checkbox. Do not select that checkbox as the client secret will be hashed and I have not yet worked on a fix for that. It turns out that when hashed, the convert-token response is invalid.

Note that, this new version of drf-social-oauth2 does not require the client_secret to be passed on the HTTPs requests.

Please, read the *Integrating Social Backends* in order to integrate social libraries into your project.

1.3 Integrating Social Backends

For each authentication provider, the top portion of your REST API settings.py file should look like this:

```
INSTALLED_APPS = (
    ...
    # OAuth
    'oauth2_provider',
    'social_django',
    'drf_social_oauth2',
)

TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                # OAuth
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
        },
    }
]

REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        # OAuth
        # 'oauth2_provider.ext.rest_framework.OAuth2Authentication',  # django-oauth-
↪toolkit < 1.0.0
        'oauth2_provider.contrib.rest_framework.OAuth2Authentication',  # django-oauth-
↪toolkit >= 1.0.0
        'drf_social_oauth2.authentication.SocialAuthentication',
    )
}
```

Listed below are a few examples of supported backends that can be used for social authentication.

For each integration for every single backend, you need to add a new application for each corresponding social backend. See the *Setting Up a New Application* section. This means that if you are authenticating with Facebook and Google, you have to create two applications in the Application section in your Django Admin dashboard.

1.3.1 Facebook Integration

To use Facebook as the authorization backend of your REST API, your settings.py file should look like this:

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Google, OpenId, etc)
    ...

    # Facebook OAuth2
    'social_core.backends.facebook.FacebookAppOAuth2',
    'social_core.backends.facebook.FacebookOAuth2',

    # drf_social_oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',

    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# Facebook configuration
SOCIAL_AUTH_FACEBOOK_KEY = '<your app id goes here>'
SOCIAL_AUTH_FACEBOOK_SECRET = '<your app secret goes here>'

# Define SOCIAL_AUTH_FACEBOOK_SCOPE to get extra permissions from Facebook.
# Email is not sent by default, to get it, you must request the email permission.
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS = {
    'fields': 'id, name, email'
}
```

To test your REST API's settings, you can execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<client_id>&backend=facebook&token=
↪<facebook_token>" http://uri:port/auth/convert-token
```

This command will return an *access_token* that you should use for every HTTP request to your API. The purpose of this process is to convert a third-party access token (*user_access_token*) into an access token that you can use with your API and its clients (*access_token*). By doing so, you will be able to authenticate each request and avoid authenticating with Facebook every time.

You can obtain the ID (*SOCIAL_AUTH_FACEBOOK_KEY*) and secret (*SOCIAL_AUTH_FACEBOOK_SECRET*) of your app from https://developers.facebook.com/apps/.

For testing purposes, you can utilize the access token *user_access_token* from https://developers.facebook.com/tools/accesstoken/.

If you require further information on how to configure python-social-auth with Facebook, visit http://python-social-auth.readthedocs.io/en/latest/backends/facebook.html.

1.3.2 Google Integration

To use Google OAuth2 as the authorization backend of your REST API, your settings.py file should look like this:

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Facebook, OpenId, etc)
    ...
    # Google  OAuth2
    'social_core.backends.google.GoogleOAuth2',
    # drf-social-oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',
    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# Google configuration
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = <your app id goes here>
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = <your app secret goes here>

# Define SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE to get extra permissions from Google.
SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE = [
    'https://www.googleapis.com/auth/userinfo.email',
    'https://www.googleapis.com/auth/userinfo.profile',
]
```

To test the configuration settings, execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<django-oauth-generated-client_id>&
↪backend=google-oauth2&token=<google_token>" http://uri:port/auth/convert-token
```

Upon successful execution, the above command returns an *access_token* that you must utilize for each HTTP request made to your REST API. In essence, what is happening here is that you are converting a third-party access token (*user_access_token*) into an access token that can be used with your API and its clients (*access_token*). For each subsequent communication between your system/application and your API, it is necessary to use this token to authenticate each request, thereby avoiding the need to authenticate with Google every time.

To obtain your app's ID (*SOCIAL_AUTH_GOOGLE_OAUTH2_KEY*) and secret (*SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET*), visit https://console.developers.google.com/apis/credentials. For more details on how to create an ID and secret, visit https://developers.google.com/identity/protocols/OAuth2.

For testing purposes, you can use the access token *user_access_token* from https://developers.google.com/oauthplayground/ and follow these steps:

- Visit the OAuth 2.0 Playground
- Select Google OAuth2 API v2 and authorize for https://www.googleapis.com/auth/userinfo.email and https://www.googleapis.com/auth/userinfo.profile
- Exchange Authorization code for tokens and get access token
- Use the access token as the token parameter in the /convert-token endpoint.

For more information on configuring python-social-auth with Google, please visit https://python-social-auth.readthedocs.io/en/latest/backends/google.html#google-oauth2.

Should you prefer a step-by-step tutorial, refer to this link provided by @djangokatya: https://djangokatya.com/2021/04/09/social-login-for-django-rest-framefork-for-newbies-a-k-a-for-me/.

1.3.3 Google OpenID Integration

OpenID and access tokens are two different concepts that are used in authentication and authorization systems.

OpenID is an open standard that allows users to authenticate with multiple websites and applications using a single set of credentials. When a user logs in using OpenID, they are redirected to their OpenID provider, which authenticates them and provides the website or application with a unique identifier for the user. The identifier can be used to retrieve the user's profile information, but it does not provide any authorization to access APIs or services.

Access tokens, on the other hand, are used to authorize API requests on behalf of the user. When a user logs in and grants permission to access their data, an access token is generated and returned to the client application. The access token is used to authenticate the client application and authorize it to make API requests on behalf of the user. The access token contains information such as the permissions granted to the client application, the expiration time, and a signature that verifies the token's authenticity.

In summary, OpenID is used to authenticate users and provide a unique identifier for them, while access tokens are used to authorize API requests on behalf of the user. While OpenID and access tokens are both important components of authentication and authorization systems, they serve different purposes and should not be confused with each other.

In order to authenticate with Open ID, proceed as follows:

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Facebook, OpenId, etc)
    ...
    # Google  OAuth2
    'drf_social_oauth2.backends.GoogleIdentityBackend',
    # drf-social-oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',
    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# Google configuration
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = <your app id goes here>
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = <your app secret goes here>

# Define SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE to get extra permissions from Google.
SOCIAL_AUTH_GOOGLE_OAUTH2_SCOPE = [
    'https://www.googleapis.com/auth/userinfo.email',
    'https://www.googleapis.com/auth/userinfo.profile',
]
```

For testing purposes, you can use the id token *id_token* from https://developers.google.com/oauthplayground/.

1. Visit the OAuth 2.0 Playground.
2. Select Google OAuth2 API v2 and authorize for openid.
3. Exchange Authorization code for tokens and get access token.
4. Use the access token as the token parameter in the /convert-token endpoint.

If you want to have your open id token validated, copy it and hit this url, https://oauth2.googleapis.com/tokeninfo?id_token=your_token_here.

To test the configuration settings, execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<django-oauth-generated-client_id>&
↪backend=google-identity&token=<google_openid_token>" http://uri:port/auth/convert-token
```

1.3.4 Github Integration

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Facebook, OpenId, etc)
    ...

    # GitHub OAuth2
    'social_core.backends.github.GithubOAuth2',

    # drf-social-oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',

    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# GitHub configuration
SOCIAL_AUTH_GITHUB_KEY = <your app id goes here>
SOCIAL_AUTH_GITHUB_SECRET = <your app secret goes here>
```

You need to register a new GitHub app at https://github.com/settings/applications/new. set the callback URL to http://example.com/complete/github/ replacing example.com with your domain.

The Client ID should be added on SOCIAL_AUTH_GITHUB_KEY and the *SOCIAL_AUTH_GITHUB_KEY* should be added on *SOCIAL_AUTH_GITHUB_SECRET*.

As described by GitHub's documentation, you need to follow a few steps in order to generate the access token to post requests on behalf of a user, team or organisation. The first step, your application will need to Request a user's GitHub identity by sending a GET request to

```
https://github.com/login/oauth/authorize
```

The only compulsory parameters are *client_id=<the app client id>* and *redirect_uri=<the redirect you added in your app>*. You will be redirected to a new location in your browser, such as http://example.com/complete/github?code=d9ba2b356d27455970bf, copy the *code=value* from it. Remember, this is only value for 10 minutes. This process should be automated by the module/library integrated in your front end application.

The second step is to send a request to:

```
$ curl -X POST -d "client_id=<client id>&client_secret=<client secret>&code=<code from␣
↪previous step>&redirect_uri=<your redirect uri>" https://github.com/login/oauth/access_
↪token
```

You should receive an access token from the previous step. Once you have the access token, test your configuration

Now, visit https://github.com/settings/tokens and create a new token. Select the user checkbox, as to grant user access. The click on the Generate Token button. Use the access token as the token parameter in the /convert-token endpoint.

To test the configuration settings, execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<django-oauth-generated-client_id>&
↪backend=github&token=<github_token>" http://uri:port/auth/convert-token
```

Read more about GitHub's configuration at Python Social Auth - Github Page

1.3.5 Instagram Integration

Before setting up any configuration in your settings.py file, you need to create an application in your Meta For Developers dashboard. Follow these guidelines in order to create and configure your application. The steps are easy to follow. Proceed until step 6.

Configure your settings.py as follows:

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Facebook, OpenId, etc)
    ...

    # Instagram OAuth2
    'social_core.backends.instagram.InstagramOAuth2',

    # drf-social-oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',

    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# Instagram configuration
SOCIAL_AUTH_INSTAGRAM_KEY = <your app id goes here>
SOCIAL_AUTH_INSTAGRAM_SECRET = <your app secret goes here>
SOCIAL_AUTH_INSTAGRAM_AUTH_EXTRA_ARGUMENTS = {'scope': 'likes comments relationships'}
```

Once you finished setting up the configuration in your project, copy the access token generated at step 5 (from facebook guidelines). Step 5 will return a response as follows:

```
{
  "access_token": "IGQVJ...",
  "user_id": 17841405793187218
}
```

Copy the access token and use it in the *token* parameter in your /auth/convert-token endpoint. To test the configuration settings, execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<django-oauth-generated-client_id>&
↪backend=instagram&token=<access_token>" http://uri:port/auth/convert-token
```

1.3.6 LinkedIn Integration

Before setting up any configuration in your settings.py file, you need to create an application in your Linked Developers dashboard. Visit the Linked Developer Page in order to create and configure your application.

For more information on how to create a new app, visit these guidelines Once your app is created, visit the Token Generator page and create a token.

Copy the access token and use it in the *token* parameter in your /auth/convert-token endpoint. To test the configuration settings, execute the following command:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<django-oauth-generated-client_id>&
↪backend=linkedin-openidconnect&token=<access_token>" http://uri:port/auth/convert-token
```

Note that: Linked In has discontinued Oauth2 since August 2023. They are using OpenID Connect instead.

Configure your settings.py as follows:

```
AUTHENTICATION_BACKENDS = (
    # Others auth providers (e.g. Facebook, OpenId, etc)
    ...

    # Linked OpenID
    'drf_social_oauth2.backends.LinkedInOpenIDUserInfo',

    # drf-social-oauth2
    'drf_social_oauth2.backends.DjangoOAuth2',

    # Django
    'django.contrib.auth.backends.ModelBackend',
)

# Instagram configuration
SOCIAL_AUTH_LINKEDIN_OPENIDCONNECT_KEY = 'key goes here'
SOCIAL_AUTH_LINKEDIN_OPENIDCONNECT_SECRET = 'secret goes here'
```

1.3.7 Other Backend Integration

DRF-Social-Oauth2 is not only limited to Google, Facebook, Instagram, Github and LinkedIn. You can integrate with every backend described at the Python Social Oauth backend integrations.

1.4 Authenticating Requests

One of the notable features of our framework is the default authentication backend, aptly named SocialAuthentication. This backend facilitates a streamlined process of user registration and authentication with your REST API.

The class functions by retrieving the backend name and token from the Authorization header, and subsequently authenticating the user through the relevant external provider. In the event that the user has not been previously registered on your app, the backend creates a new user for this purpose, ensuring a seamless authentication process.

1.4.1 Authentication Ready View

You can set up a view which requires authentication just by inheriting from the generics class of Django Rest Framework, as shown below:

```
from rest_framework import generics

class MyView(generics.ListAPIView):
def get(self, request, *args, **kwargs):
    response = {
        'message': 'token works.'
    }
    return Response(response, status=200)
```

If, by any chance you need a view without authentication, just set the *authentication_class*

```
from rest_framework.permissions import AllowAny

class MyView(generics.ListAPIView):
authentication_classes = (AllowAny,)

def get(self, request, *args, **kwargs):
    response = {
        'message': 'token works.'
    }
    return Response(response, status=200)
```

Include the header *Authorization* to request, and your view should respond if your access token is valid:

```
$ curl -H "Authorization: Bearer <backend_name> <backend_token>" http://localhost:8000/
↪route/to/your/view
```

1.5 Running local tests

The unit tests for drf-social-oauth2 are located in the tests/ directory. To run these tests on your local machine, you will need to first build the Docker image and then execute the test run command. To do so, follow these steps:

Build the Docker image by running the following command:

```
$ docker-compose -f docker-compose.tests.yml build --no-cache
```

Execute the test run command by running the following command:

```
$ docker-compose -f docker-compose.tests.yml up --exit-code-from app
```

Once the tests have completed, you can view the results in the htmlcov/ folder in your local environment. The index.html file provides detailed information about the test coverage of the project.

To clean up your local system and remove all containers created during the testing process, run the following command:

```
$ docker-compose -f docker-compose.tests.yml down
```

1.6 Further Customization for Drf-Social-Oauth2

This section is meant for further customization of the framework. Any idea is welcome and will be listed here.

1.6.1 Customize token expiration

To customize the expiration time for tokens, you can easily do so by adjusting the settings in your *settings.py* file.

Simply import the *oauth2_provider* settings and set the *ACCESS_TOKEN_EXPIRE_SECONDS* to your desired value, in seconds.

Here's an example of how to set the expiration time to 6 months:

```
# in your settings.py file.
from oauth2_provider import settings as oauth2_settings

# expires in 6 months
oauth2_settings.DEFAULTS['ACCESS_TOKEN_EXPIRE_SECONDS'] = 1.577e7
```

By customizing the token expiration time, you can fine-tune the security and functionality of your application to suit your specific needs.

1.6.2 Refresh Token Rotation

Refresh token rotation is a security feature that issues a new refresh token each time a refresh token is used to obtain a new access token. This helps mitigate the risk of refresh token theft.

How it works:

1. When a client uses a refresh token to get a new access token, a new refresh token is also issued.
2. The old refresh token is invalidated and cannot be used again.
3. If someone tries to reuse an old refresh token (indicating potential theft), all tokens in that "family" are revoked.

Configuration:

Add the following to your `OAUTH2_PROVIDER` settings in `settings.py`:

```
OAUTH2_PROVIDER = {
    # Enable refresh token rotation (default: True)
    'ROTATE_REFRESH_TOKEN': True,

    # Enable reuse protection - revokes all tokens if a used refresh token
    # is reused (default: True)
    'REFRESH_TOKEN_REUSE_PROTECTION': True,

    # Grace period in seconds - how long the old refresh token remains
    # valid after rotation to handle concurrent requests (default: 0)
    'REFRESH_TOKEN_GRACE_PERIOD_SECONDS': 30,

    # Refresh token lifetime in seconds (default: 14 days)
    'REFRESH_TOKEN_EXPIRE_SECONDS': 1209600,

    # Access token lifetime in seconds (default: 1 hour)
    'ACCESS_TOKEN_EXPIRE_SECONDS': 3600,
}
```

Settings explained:

- `ROTATE_REFRESH_TOKEN`: When `True`, a new refresh token is issued each time a refresh token is used.
- `REFRESH_TOKEN_REUSE_PROTECTION`: When `True`, if a refresh token that has already been used is used again, all refresh tokens for that user/application are revoked. This protects against token theft.
- `REFRESH_TOKEN_GRACE_PERIOD_SECONDS`: The number of seconds the old refresh token remains valid after rotation. This helps handle race conditions when multiple requests use the same refresh token simultaneously.
- `REFRESH_TOKEN_EXPIRE_SECONDS`: The absolute lifetime of refresh tokens in seconds.

Client-side considerations:

When refresh token rotation is enabled, your client application must:

1. Store the new refresh token returned with each token refresh response.
2. Replace the old refresh token with the new one.
3. Handle the case where a refresh fails due to token reuse (re-authenticate the user).

Example refresh response:

```
{
    "access_token": "new_access_token_here",
    "refresh_token": "new_refresh_token_here",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "read write"
}
```

Security benefits:

- **Limits token lifetime**: Even if a refresh token is stolen, it can only be used once.
- **Detects theft**: If an attacker uses a stolen refresh token after the legitimate user has already used it, the reuse is detected and all tokens are revoked.
- **Reduces attack window**: The grace period can be set to a small value to minimize the window of vulnerability.

1.6.3 Google Email Alias Normalization

Google treats `@gmail.com` and `@googlemail.com` as aliases of the same mailbox — the `@googlemail.com` form is still used in some regions (e.g. Germany, the UK). However, `social_core`'s default Google backends derive the social UID from the email address, so the same Google user signing in once with `foo@gmail.com` and once with `foo@googlemail.com` would otherwise produce two distinct `UserSocialAuth` records and two Django users.

drf-social-oauth2 normalizes `@googlemail.com` to `@gmail.com` before the UID is computed, so both forms map to the same Django user.

Automatic for GoogleIdentityBackend:

If you authenticate with `drf_social_oauth2.backends.GoogleIdentityBackend` (the OpenID Connect backend recommended in the *Integrating Social Backends* guide), normalization is applied automatically — no extra configuration is required.

Pipeline step for other Google backends:

If you use a stock `python-social-auth` Google backend such as `social_core.backends.google.GoogleOAuth2`, add the `drf_social_oauth2.pipeline.normalize_google_email` step to your `SOCIAL_AUTH_PIPELINE`, **before** `social_core.pipeline.social_auth.social_uid`:

```
SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    # Normalize @googlemail.com -> @gmail.com before the UID is computed.
    'drf_social_oauth2.pipeline.normalize_google_email',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
```

(continues on next page)

(continued from previous page)

```
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

The step is a no-op for non-Google backends (matched by a backend name starting with `google`) and for emails that don't use the `@googlemail.com` alias, so it is safe to leave enabled in mixed-provider deployments.

1.7 Testing the Setup

Welcome to the installation guide. Now that you have completed the installation, let's explore the various functionalities provided by this package. For the following examples, we will assume that the REST API is reachable at http://localhost:8000.

To retrieve a token for a user, you can use the following command with curl:

```
$ curl -X POST -d "client_id=<client_id>&client_secret=<client_secret>&grant_
↪type=password&username=<user_name>&password=<password>" http://localhost:8000/auth/
↪token
```

Here, replace *client_id* and *client_secret* with the keys generated automatically by the Application model you created.

To refresh a token, use the following command:

```
$ curl -X POST -d "grant_type=refresh_token&client_id=<client_id>&client_secret=<client_
↪secret>&refresh_token=<your_refresh_token>" http://localhost:8000/auth/token
```

You can exchange an external token for a token linked to your app using:

```
$ curl -X POST -d "grant_type=convert_token&client_id=<client_id>&client_secret=<client_
↪secret>&backend=<backend>&token=<backend_token>" http://localhost:8000/auth/convert-
↪token
```

Here, replace *backend* with the name of an enabled backend and *backend_token* with the token you received from the external service.

Finally, to revoke tokens, use the following commands:

To revoke a single token:

```
$ curl -X POST -d "client_id=<client_id>&client_secret=<client_secret>&token=<your_token>
↪" http://localhost:8000/auth/revoke-token
```

To revoke all tokens for a user:

```
$ curl -H "Authorization: Bearer <token>" -X POST -d "client_id=<client_id>" http://
↪localhost:8000/auth/invalidate-sessions
```

To revoke only refresh tokens:

```
$ curl -H "Authorization: Bearer <token>" -X POST -d "client_id=<client_id>" http://
↪localhost:8000/auth/invalidate-refresh-tokens
```

No need to build your own request as you can also use the provided curl commands or the Swagger interface.

1.8 OpenAPI Specs

To interact with your API and make requests, you can utilize the Swagger Editor. The following commands will enable you to run the Swagger Editor and begin interacting with your API.

For Mac and Linux users, run the following command:

```
$ docker run --rm -p 8080:8080 -v $(pwd):/tmp -e SWAGGER_FILE=/tmp/api.yaml swaggerapi/
↪swagger-editor
```

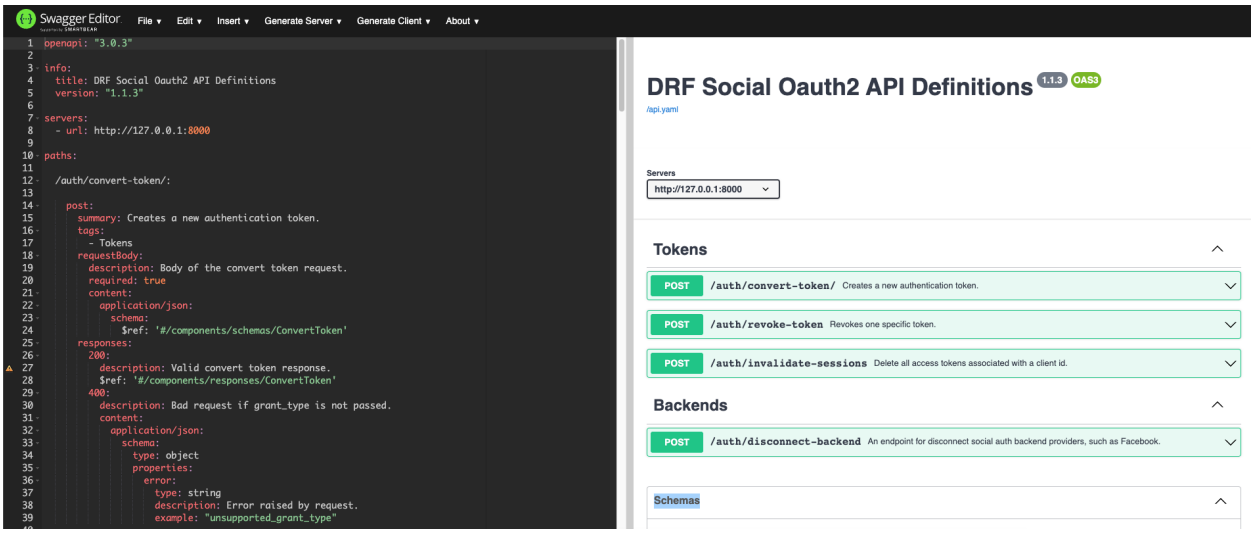
For Windows users, run the following command:

```
$ docker run --rm -p 8080:8080 -v ${pwd}:/tmp -e SWAGGER_FILE=/tmp/api.yaml swaggerapi/
↪swagger-editor
```

You don't need to build your own requests from scratch, as both curl commands and the Swagger interface are provided. With these tools at your disposal, you can easily interact with your API and test its functionality.

You can access the Swagger application by visiting http://localhost:8080.

The Swagger console looks like this:



CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search