OAuth2 in Python

A practical guide to OAuth2 internals for Python programmers, with examples for GitHub and Facebook.

Written by: Neven Munđar <neven.mundar@dobarkod.hr>

Last change: February 10th, 2014

GoodCode

http://www.goodcode.io/

In first part of this guide, we'll explain general behavior of OAuth 2.0 authorization protocol and the approach a developer might take when writing application that uses it.

Developers usually encounter OAuth 2.0 protocol when they need to develop software that interacts with API of GitHub, Google, Facebook or some similar web service. Those services often have quite good documentation that explains how to implement software that can interact with them.

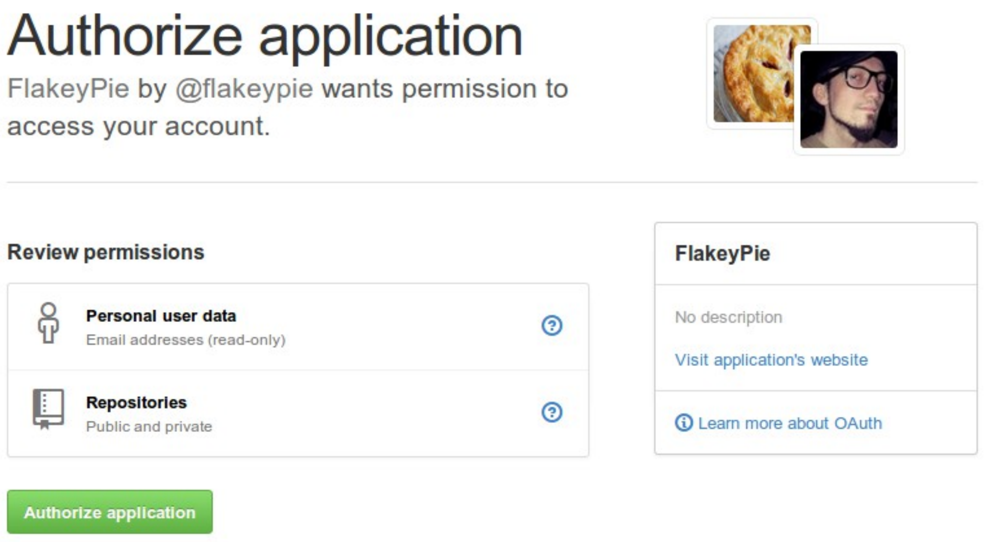
Implementing OAuth protocol flow is not something that entertains a lot of people. With a bit of luck you can find a couple of quite good libraries for popular languages that can get the job done. Python developers can use a library like requests-oauthlib, or framework-specific solution like django-allauth or go with python-social-auth, a library that provides support for multiple python web frameworks. After all, many people just need an access token and a library that hides the gritty details of the API calls and they're set. Picking a well supported library means trusting a person or a team to stay up to date with particular standard, be quick on updates and handle issues quickly and painlessly. You can usually check the rate of updates, current issues and developers public responses to issues and make an informed guess about the quality of the library. Finding the right library that can handle OAuth 2.0 is not such a difficult task because it's a solid standard and a need for intervention rarely arises, usually when a web service that you use changes the way it operates.

Handling OAuth by yourself

But sometimes a person wants a clarification of some process that is happening. Documentation of a web service and library will explain HOW to perform authorization, but explaining why are some details needed would just unnecessarily prolong required time to get the job done. We'll go through Python examples of OAuth 2.0 authorization flow for GitHub and Facebook services using Requests library that can handle HTTP calls quite elegantly and Django web framework (any other similar web framework is equally capable for these examples) to setup our HTTP endpoints.

OAuth 2.0 is authorization standard whose purpose is to provide a way to access restricted resources over HTTP. Instead of using username and password of resource owner to access some resource, client application can obtain an access token. With access token, third-party applications don't need to store owner's username and password. Also, access to resources can be time limited and scope limited so that clients don't gain overly broad access and resource owners can revoke access to individual clients instead of changing their credentials.

For example, GitHub user (resource owner) can give permission to application (client) to access his private repositories (resources) without sharing his username/password. When user is requested to authorize the application he is shown a list of requested permissions that he can accept or deny. Authorized client can then authenticate directly with GitHub authorization server to get the access token.



Authorize application

FlakeyPie by @flakeypie wants permission to access your account.

Review permissions

Personal user data
Email addresses (read-only)

Repositories
Public and private

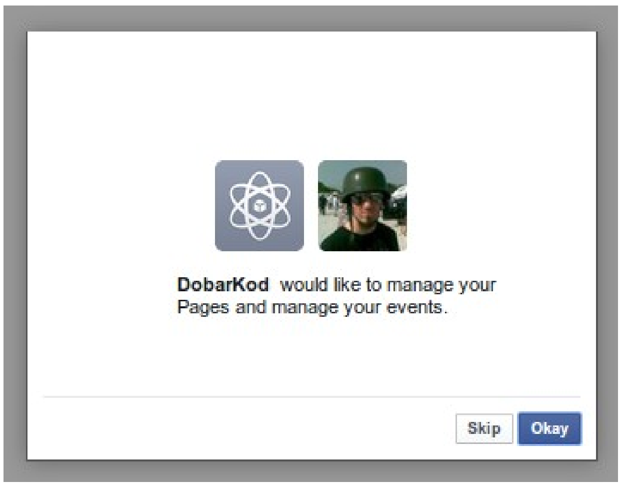
FlakeyPie

No description

Visit application's website

Learn more about OAuth

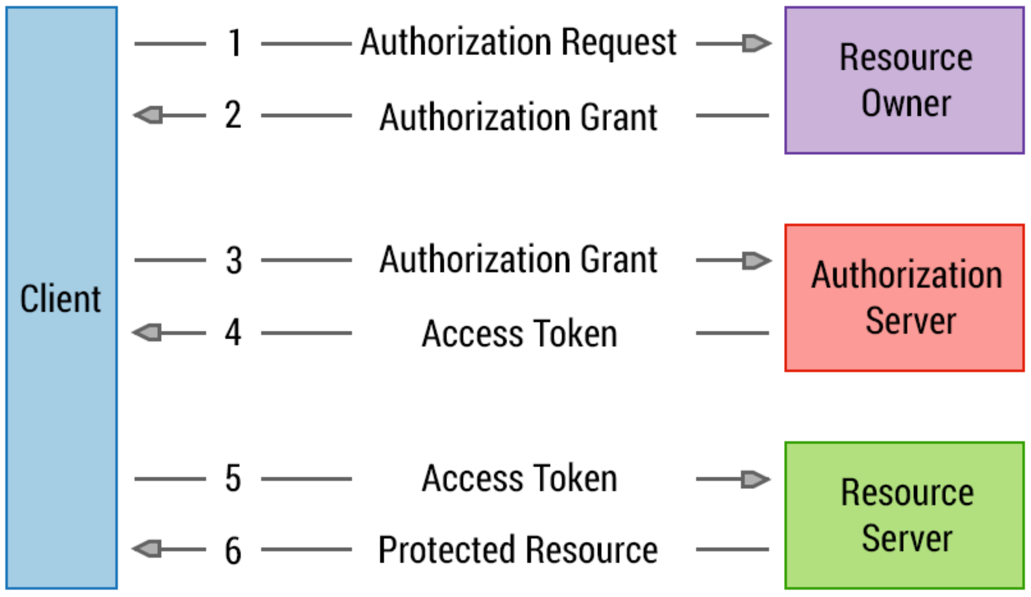
Likewise, Facebook user (resource owner) can grant access to his wall or pages (resources) to application (client) without sharing his username/password. Facebook users are shown each permission separately and they can choose to skip the ones they don't like.



DobarKod would like to manage your Pages and manage your events.

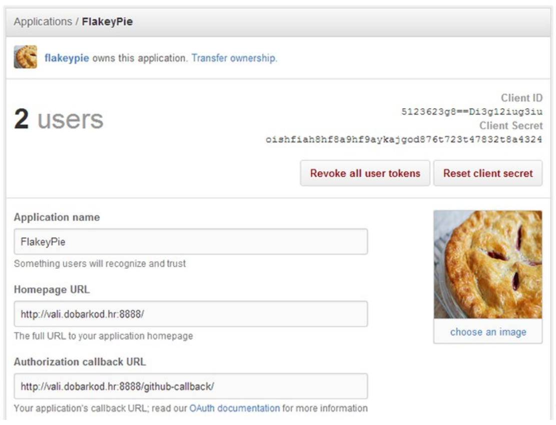
OAuth Flow

Process of authorizing your application for access to users resources using OAuth 2.0 protocol is called OAuth flow. The flow for getting and using access token consists of 6 steps. First, client application requests rights to access users resources (1). If the user chooses to grant the rights (2) client can use that grant to request valid access token from authorization server (3). If authorization server validates the grant client will be issued access token (4). Client can then request the users resources on the resource server using access token for authentication (5). Resource server then serves the resources if the token is valid (6).



Application registration

First step in our OAuth adventures is registering the application with GitHub and Facebook service. Besides naming the app we'll have to provide URL of the service. GitHub needs complete URL where authorization responses will be sent. Facebook only needs the base URL for validation of the domain from where requests originate. Completing the application registration will give us a set of credentials to use for authentication with authorization server. For GitHub service that will be Client ID/Client Secret pair which Facebook calls the App ID/App Secret.

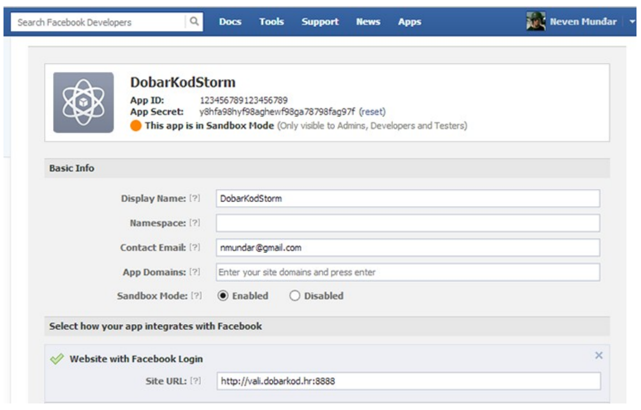


Client ID
5123623g8==Di3g12iug3iu
Client Secret
oishfiah8hf8a9hf9aykajgod876t723t47832t8a4324

Application name
FlakeyPie
Something users will recognize and trust

Homepage URL
http://vali.dobarkod.hr:8888/
The full URL to your application homepage

Authorization callback URL
http://vali.dobarkod.hr:8888/github-callback/
Your application's callback URL; read our OAuth documentation for more information



DobarKodStorm
App ID: 123456789123456789
App Secret: y8hfa98hyf98aghewf98ga78798fag97f (reset)
This app is in Sandbox Mode (Only visible to Admins, Developers and Testers)

Basic Info
Display Name: [?] DobarKodStorm
Namespace: [?]
Contact Email: [?] nmundar@gmail.com
App Domains: [?] Enter your site domains and press enter
Sandbox Mode: [?] [✓ Enabled] Disabled

Select how your app integrates with Facebook
✓ Website with Facebook Login
Site URL: [?] http://vali.dobarkod.hr:8888

Access token lifetime

Access tokens usually have limited time period during which they are active. In case of Facebook this is hour or two for short lived tokens and about 60 days for long lived tokens. Facebook does not provide any way to automatically get new token after expiration. This results in users having to go through Facebook login procedure again when token expires, but they won't have to go through the permissions approval again. In the past Facebook provided offline access token that could be extended without asking the user, but this is no longer the case.

On the other hand, access tokens issued by GitHub don't have set expiration time. We can request new access token frequently, for example each time user logs in, or keep the token in database and avoid hitting the GitHub API if we already have the token. Since requesting the token is the main point of this article, for simplicity's sake we'll get the token each time user logs in.

GitHub Login

Finally, time to do some coding. Let's enable users to login to our Django site with their GitHub account and then we can perform some actions using GitHub API. We won't go into too much detail on how to perform Django login since official docs cover that well. Django will just serve as a medium for our OAuth flow deliciousness.

Authorization Request

Let's ask the user for a right to access his repositories on GitHub. Using Django we can create a view that will start the OAuth flow by defining the scope of permissions and redirecting the user to GitHub server to let him choose to allow or deny access to your app. We can send a user to this view via button or link.

urls.py

```python
from django.conf.urls import patterns, url

urlpatterns = patterns('accounts.views',
    # ...
    url(r'^github-connect/$', 'github_connect', {}, 'github_connect')
)
```

views.py

```python
import uuid
from django.shortcuts import redirect
from requests import Request

def github_connect(request):
    state = uuid.uuid4().get_hex()
    request.session['github_auth_state'] = state
    params = {
        'client_id': '123asd45678a12345678',
        'scope': 'user:email, repo',
        'state': state
    }
    github_auth_url = 'https://github.com/login/oauth/authorize'
    r = Request('GET', url=github_auth_url, params=params).prepare()
    return redirect(r.url)
```

We will create a **state** variable which will be a long hexadecimal string. It's purpose is CSRF protection (this enables us to make sure incoming responses actually originate from trusted services). Subsequent responses from service will contain that state string which must match with our saved state. State will be kept in session and compared with the state value that GitHub returns with authorization grant.

A list of permissions that we request will be send as **scope** parameter. For a list of valid scopes consult GitHub API docs. We specifically want users email and repository access. This permission will serve our diabolical intentions to spell check users repositories and mail them the reports.

In addition to state and scope we must send our **Client ID** that we got when we registered our application with GitHub. This ID uniquely identifies our application to authorization server. Scope is a string of requested permissions.

User is then sent to GitHub service and shown requested permissions and given options to allow or deny the request to access his resources. If he accepts the request our app will be issued authorization grant. This grant comes in the form of "**code**" GET parameter as part of the url that redirects users back to your site. This code has a very short lifespan (up to 10 minutes) and can be used only once to request access token from authorization server. After giving us the grant user will be returned to our website. Return URL will be the one configured in the OAuth application settings. It's possible to give a different url with redirect_uri parameter, but it's entirely optional.

Receiving the Authorization Grant

To receive the grant we must construct another endpoint on our server. This is the endpoint that we've configured in our application settings and is used by authorization server to send us the authorization grant.

urls.py

```
from django.conf.urls import patterns, url
urlpatterns = patterns('accounts.views',
    # ...
    url(r'^github-callback/$', 'github_callback', {}, 'github_callback')
)
```

views.py

```
from django.http import Http404

def github_callback(request):
    original_state = request.session.get('github_auth_state')
    if not original_state:
        raise Http404
    del(request.session['github_auth_state'])

    state = request.GET.get('state')
    code = request.GET.get('code')

    if not state or not code:
        raise Http404
    if original_state != state:
        raise Http404
    # request token...
```

When we get a request from GitHub server we will compare received state with the original and check for the existence of **code** that contains users grant. In case of problems we can just raise 404 error and ignore the request for simplicity's sake.

Getting the Access Token

With received code we can finally request the access token. In order to do that we must send our GitHub Client ID and GitHub Client Secret along with the received **code** via POST request to GitHub's access token endpoint.

```
import requests

def github_callback(request):
    #  handle grant...
    params = {
        'client_id': '123asd45678a12345678',
        'client_secret': '12345qwert12345qwert12345qwert12345qwert',
        'code': code
    }
    headers = {'accept': 'application/json'}
    url = 'https://github.com/login/oauth/access_token'
    r = requests.post(url, params=params, headers=headers)

    if not r.ok:
        raise Http404

    data = r.json()
    access_token = data['access_token']
    #  ...
```

If everything went fine, we will receive **access token** in a JSON response (since we requested JSON response in accept header). From now on we can sign our requests to different GitHub API endpoints with this token and we will receive user's resources as long as they're included in the scope of permissions that they gave us.

Using Access Token

So, what can we do now that we have the OAuth holy handgrenade in form of access token? We can send HTTP requests to GitHub API endpoints. For example, lets use requests library to get the information about the user whose access token we just got. We need to create authorized GET request on ***/user*** endpoint. When we make the call we can send the token as a GET parameter or as a part of the Authorization header.

```
import requests

headers = {
    'authorization': 'token %s' %  access_token
}
r = requests.get('https://api.github.com/user', headers=headers)

r.json()
>>> {
    'id': 1196939,
    'type': 'User',
    'login': 'nmundar',
    'url': 'https://api.github.com/users/nmundar',
    'public_repos': 3,
    'repos_url': 'https://api.github.com/users/nmundar/repos',
    'avatar_url': 'https://gravatar.com/avatar/54abfc8ede828...',
    # other stuff...
}
```

We made the call with access token as part of the Authorization header. We received lots of info about the user and we can use it to create a user in our database, log him in or call any GitHub API endpoint for which the user gave us permissions.

Since taking care of many similar API calls can be quite cumbersome for more than a few calls I'd like to recommend pretty good Python library for GitHub API. It's called PyGithub. It's pretty stable, intuitive and easy to use. Using PyGitHub lib to get the user info can be written like this:

```
from github import Github

user = Github(login_or_token=access_token).get_user()
user.login
>>> u'nmundar'
```

In our app we can complete the callback function by creating new instance of our User model called GitHubUser model for the user that just completed the flow. This is where the details of your environment come into play, and we will use Django's authentication and login functionality.

models.py

```
from github import Github
from django.shortcuts import redirect
from django.contrib.auth import authenticate
from django.contrib.auth import login
from .models import GitHubUser

def github_callback(request):
    #  get the token...
    g_user = Github(login_or_token=access_token).get_user()
    user = GitHubUser.objects.create_user(
        login=g_user.login, token=access_token
    )
    user = authenticate(login=user.login)
    login(request, user)
    return redirect('home')
```

In this very simple example we'll just create a new user with given credentials. In real world you'll additionally want to check if the user exist and update his credentials instead but user management is out of the scope of this guide.

With this in place we have everything we need for user authentication and GitHub API access.

Facebook

Let's move away from plain "user-logs-in-to-site-with-another-service" scenario and show a more elaborate use case. OAuth login is mainly used by websites to make it easier for general public to use their site without complicated account registration procedure. Moving away from that we can develop some pretty specific apps. For example, let's develop a Facebook bot that will serve the needs of only one user. Let's say that the user is administrator of a website and wants to automatically update his Facebook Page with details about various events that happen on his service. We can setup a simple event monitor and when the event happens we'll use Facebook API to create status updates on Facebook Page.

So we'll needed to create an app for a single user (Django admin) and make it easy for him to activate the app. We can keep all the data we need in a single database row. This will make configuration easy since we can use Django admin UI for a single model. Django singleton model will serve that purpose perfectly. We can create abstract base model first and then derive our concrete model from that.

Singleton ensures there's always only one entry in the database, and can fix the table (by deleting extra entries) even if added via another mechanism. It has a static load() method which always returns the object – from the database if possible, or a new empty (default) instance if the database is still empty.

For the purposes of this guide details of the singleton implementation are not really relevant. Example of abstract singleton model can be found here.

Facebook Credentials Singleton

We can create concrete model that will store all the data needed to make Facebook API calls:

settings.py

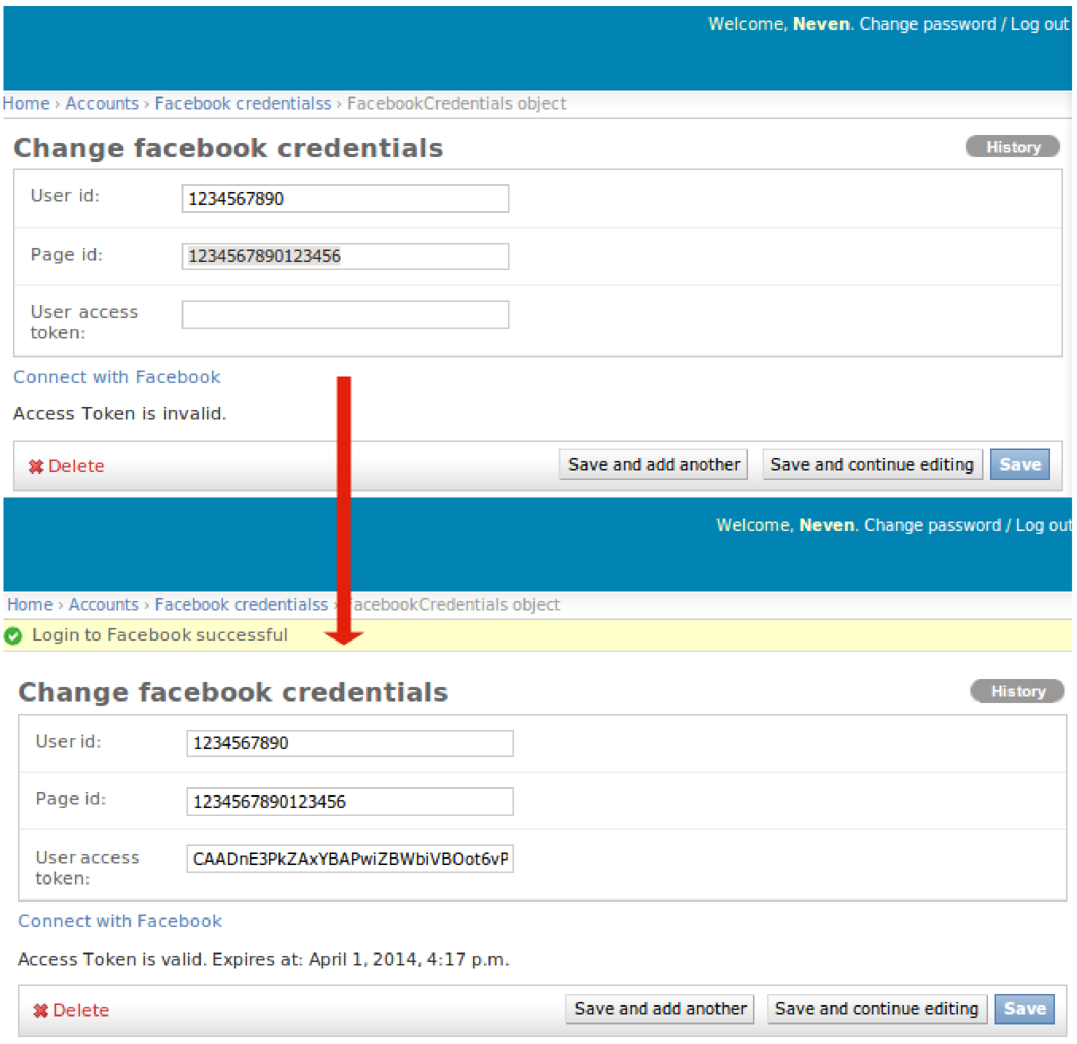
```
FACEBOOK_APP_ID = '123456789123456'
FACEBOOK_API_SECRET = '123asd456asd789asd321'
FACEBOOK_SCOPE = 'publish_stream,manage_pages,status_update,create_event'
```

models.py

```
from django.db import models

class FacebookCredentials(SingletonModel):
    FACEBOOK_APP_ID = settings.FACEBOOK_APP_ID
    FACEBOOK_API_SECRET = settings.FACEBOOK_API_SECRET
    FACEBOOK_SCOPE = settings.FACEBOOK_SCOPE
    FACEBOOK_REDIRECT_URL = settings.BASE_URL + '/accounts/facebook-callback/'
    TOKEN_RE = 'access_token=(?P<access_token>[^&]+)(?:&expires=(?P<expires>.*))?'

    facebook_url = 'https://www.facebook.com'
    graph_url = 'https://graph.facebook.com'
    access_token_url = graph_url + '/oauth/access_token'
    oauth_url = facebook_url + '/dialog/oauth'
    token_debug_url = graph_url + '/debug_token'

    user_id = models.CharField(max_length=256, blank=True, null=True)
    page_id = models.CharField(max_length=256, blank=True, null=True)
    user_access_token = models.CharField(max_length=512, blank=True, null=True)

    # functions ...
```

We need to store user's id to access his resources on Facebook, page id for posting updates and access token for posting authorization. Besides that we'll keep all application settings in the model along with all the various urls and utility functions.

This is the desired functionality in the admin interface:


Welcome, **Neven**. Change password / Log out

Home › Accounts › Facebook credentialss › FacebookCredentials object

Change facebook credentials

User id:	1234567890
Page id:	1234567890123456
User access token:	

Connect with Facebook

Access Token is invalid.

Welcome, **Neven**. Change password / Log out

Home › Accounts › Facebook credentialss › FacebookCredentials object

Login to Facebook successful

Change facebook credentials

User id:	1234567890
Page id:	1234567890123456
User access token:	CAADnE3PkZAxYBAPwiZBWbiVBOot6vP

Connect with Facebook

Access Token is valid. Expires at: April 1, 2014, 4:17 p.m.

Admin logs in, sets his Facebook User ID and Page ID, clicks “Connect with Facebook” and goes through OAuth flow. At the end of the flow we get access token along with the info about tokens validity and then our system is ready to start using the API. “Connect with Facebook” link in admin interface will be simple URL that will redirect user to Facebook login page. Login url is then used in admin template.

Authorization Request

Lets get started with the flow. This time we'll skip the url configuration to simplify the code.

views.py

```
import uuid
from requests import Request
from django.contrib.admin.views.decorators import staff_member_required
from .models import FacebookCredentials

@staff_member_required
def facebook_connect(request):
    c = FacebookCredentials.load()
    state = uuid.uuid4().get_hex()
    request.session['facebook_auth_state'] = state
    params = {
        'client_id': c.FACEBOOK_APP_ID,
        'scope': c.FACEBOOK_SCOPE,
        'state': state,
        'redirect_uri': c.FACEBOOK_REDIRECT_URL
    }
    r = Request('GET', url=c.oauth_url, params=params).prepare()
    return redirect(r.url)
```

This is similar to GitHub example. We send our client ID, scope and state. Client ID identifies our app, scope defines the permissions that we request, and state is used for csrf protection. Difference between GitHub example and this is in parameters that we send. Here we explicitly state the return url with 'redirect_uri' parameter. Facebook requires this parameter while GitHub makes it optional.

Receiving Authorization Grant

Similar to our GitHub example, we need to setup our callback method that will receive the grant from the user and get valid access token.

views.py

```python
from django.contrib import messages

def facebook_callback(request):
    error = request.GET.get('error')
    if error:
        description = request.GET.get('description')
        messages.add_message(request, messages.ERROR, description)

    state = request.GET.get('state')
    original_state = request.session.get('facebook_auth_state')
    if state != original_state:
        description = "Unauthorized authentication action."
        messages.add_message(request, messages.ERROR, description)

    # get token...
```

This time we handle errors by showing them to the user utilizing Django messaging framework. If the user denies permission by clicking the “Cancel” button, we will receive **user_denied** as error parameter and “User denied your request” in description parameter. Next, we check if the received state parameter matches originally generated state parameter that we sent to the server. If it doesn't, there is a possibility of CSRF attack.

After this, we can extract the code parameter and get the access token. Getting the access token is bit trickier that with GitHub. If this is the first authorization of this app for this user when we get the token in request body beside it will be **expires** parameter. This means that we received short lived token (1-2 hours) and we'll need to exchange it for long lived token.

Getting the Access Token

Let's write helper method that gets the access token:

models.py

```
import re
import requests

class FacebookCredentials(SingletonModel):
    # ...
    TOKEN_RE = 'access_token=(?P<access_token>[^&]+)(?:&expires=(?P<expires>.*))?'
    # ...
    def get_user_access_token_from_code(self, code):
        payload = {
            'client_id': self.FACEBOOK_APP_ID,
            'client_secret': self.FACEBOOK_API_SECRET,
            'redirect_uri': self.FACEBOOK_REDIRECT_URL,
            'code': code,
        }
        r = requests.get(self.access_token_url, params=payload)
        data = re.compile(self.TOKEN_RE).match(r.text).groupdict()
        self.user_access_token = data['access_token']
        self.save()
```

Same as before, we send client id and secret to the Facebook server, along with the received code. The difference between Facebook and OAuth specification is that requires that we send original redirect_url as parameter. Additionally Facebook wants us to perform GET request on token endpoint, while specification requires POST, but at least Graph API is on HTTPS. So we send the data, receive the response and then extract the token from request body with a regular expression. We ignore the **expires** parameter because we will always extend the token. Access token is then saved to database.

Extending the Token

Extending the token is preformed by sending the short lived token to access token endpoint along with **grant_type** parameter set as **fb_exchange_token**:

models.py

```python
import re
import requests

class FacebookCredentials(SingletonModel):
    # ...
    TOKEN_RE = 'access_token=(?P<access_token>[^&]+)(?:&expires=(?P<expires>.*))?'
    # ...
    def extend_token(self):
        params = {
            'client_id': self.FACEBOOK_APP_ID,
            'client_secret': self.FACEBOOK_API_SECRET,
            'grant_type': 'fb_exchange_token',
            'fb_exchange_token': self.user_access_token
        }
        r = requests.get(self.access_token_url, params=params)
        data = re.compile(self.TOKEN_RE).match(r.text).groupdict()
        self.user_access_token = data['access_token']
        self.save()
```

Completing the Flow

With access token handling functions in place we can complete our OAuth flow in the callback function.

views.py

```
from django.shortcuts import render, redirect
from django.contrib import messages
from .models import FacebookCredentials

def facebook_callback(request):
    # ...check erros and state

    c = FacebookCredentials.load()
    code = request.GET.get('code')
    if code:
        c.get_user_access_token_from_code(code)
        c.extend_token()
        messages.add_message(request, messages.INFO,
            _('Login to Facebook successful'))

    return redirect('admin:accounts_facebookcredentials_change', c.id)
```

After the flow is completed we'll have the extended token in our database and we can create the usual Graph API calls. We send the user back to admin page from where he started and display him appropriate message. Adding the “Connect with Facebook” link to admin page is performed by extending Django **change_form.html** template and adding template tag for our **facebook_connect** view. With this set we're ready to go! Facebook has a pretty good tool for playing with the API called Graph API Explorer. With it you can test the API as you develop your app and explore all the various options. So now we could see how to post to Facebook pages, update statuses, create events and similar actions. Since we've gone through all the things required to do that tasks we'll leave it as an exercise to the reader.

We hope this guide has helped you demystify various bits & pieces of OAuth flow and set you on the right track.