django-oidc-provider Documentation

Release 0.5.x

Juan Ignacio Fiorentino

Jul 11, 2017

Contents

This tiny (but powerful!) package can help you providing out of the box all the endpoints, data and logic needed to add OpenID Connect capabilities to your Django projects. And as a side effect a fair implementation of OAuth2.0 too. Covers Authorization Code, Implicit and Hybrid flows.

Also implements the following specifications:

- OpenID Connect Discovery 1.0
- OpenID Connect Session Management 1.0
- OAuth 2.0 for Native Apps
- OAuth 2.0 Resource Owner Password Credentials Grant
- Proof Key for Code Exchange by OAuth Public Clients

Before getting started there are some important things that you should know:

- Despite that implementation MUST support TLS. You can make request without using SSL. There is no control on that.
- Supports only for requesting Claims using Scope values.
- If you enable the Resource Owner Password Credentials Grant, you MUST implement protection against brute force attacks on the token endpoint

---

Contents:

CHAPTER 1

Installation

Requirements

- Python: `2.7` `3.4` `3.5` `3.6`
- Django: `1.7` `1.8` `1.9` `1.10` `1.11`

Quick Installation

If you want to get started fast see our `/example_project` folder.

Install the package using pip:

```
$ pip install django-oidc-provider
```

Add it to your apps:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'oidc_provider',
    # ...
)
```

Add the provider urls:

```
urlpatterns = patterns('',
    # ...
    url(r'^openid/', include('oidc_provider.urls', namespace='oidc_provider')),
```

```
    # ...
)
```

Generate server RSA key and run migrations (if you don't):

```
$ python manage.py migrate
$ python manage.py creatersakey
```

Add required variables to your project settings:

```
LOGIN_URL = '/accounts/login/'
```

CHAPTER 2

Relying Parties

Relying Parties (RP) creation it's up to you. This is because is out of the scope in the core implementation of OIDC. So, there are different ways to create your Clients (RP). By displaying a HTML form or maybe if you have internal thrusted Clients you can create them programatically.

OAuth defines two client types, based on their ability to maintain the confidentiality of their client credentials:

- `confidential`: Clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials).
- `public`: Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

Properties

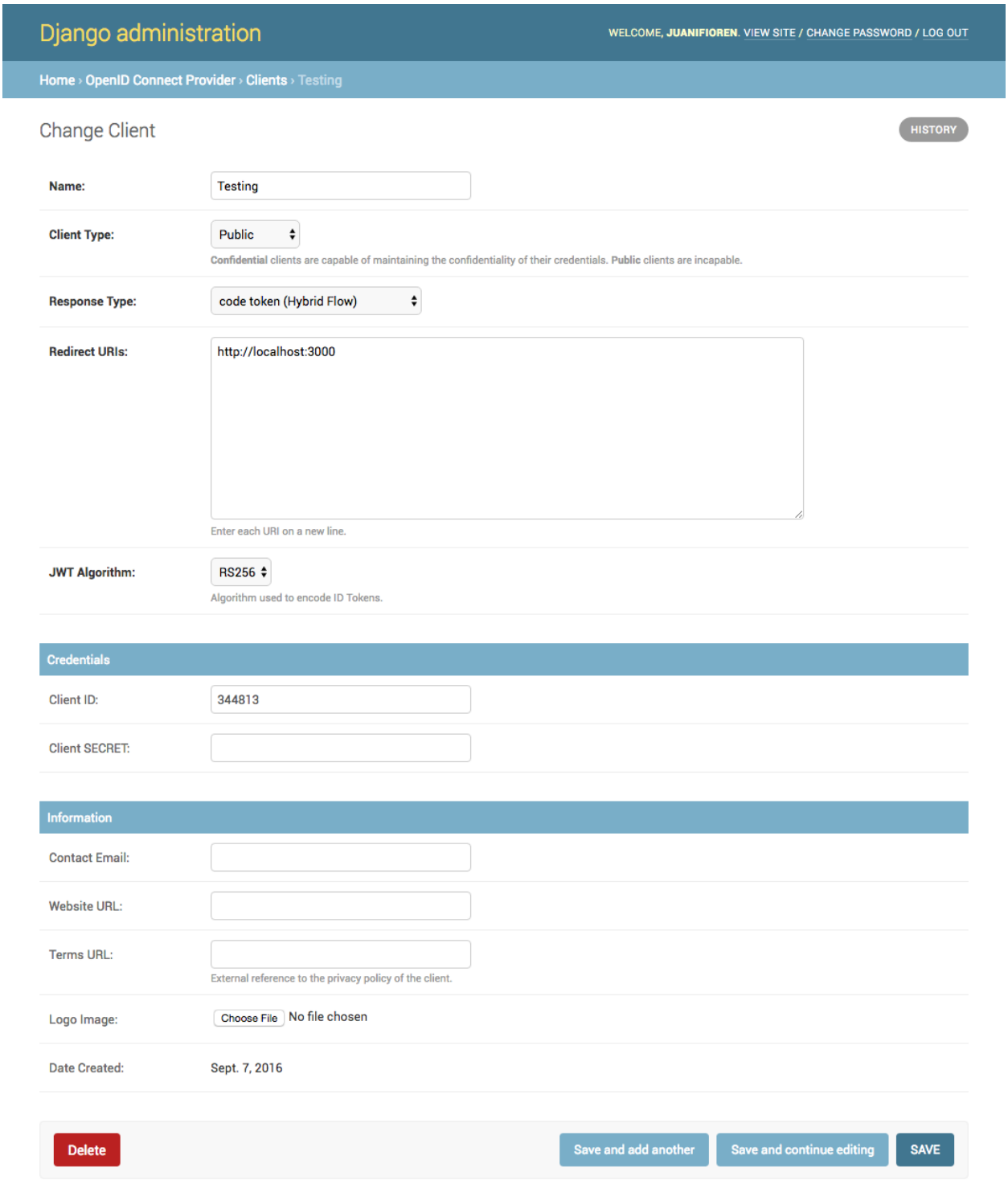
- `name`: Human-readable name for your client.
- `client_type`: Values are `confidential` and `public`.
- `client_id`: Client unique identifier.
- `client_secret`: Client secret for confidential applications.
- `response_type`: Values depends of wich flow you want use.
- `jwt_alg`: Clients can choose wich algorithm will be used to sign id_tokens. Values are `HS256` and `RS256`.
- `date_created`: Date automatically added when created.
- `redirect_uris`: List of redirect URIs.
- `require_consent`: If checked, the Server will never ask for consent (only applies to confidential clients).
- `reuse_consent`: If enabled, the Server will save the user consent given to a specific client, so that user won't be prompted for the same authorization multiple times.

Optional information:

- `website_url`: Website URL of your client.
- `terms_url`: External reference to the privacy policy of the client.
- `contact_email`: Contact email.
- `logo`: Logo image.

Using the admin

We suggest you to use Django admin to easily manage your clients:



For re-generating `client_secret`, when you are in the Client editing view, select "Client type" to be `public`. Then after saving, select back to be `confidential` and save again.

Custom view

If for some reason you need to create your own view to manage them, you can grab the form class that the admin makes use of. Located in `oidc_provider.admin.ClientForm`.

Some built-in logic that comes with it:

- Automatic `client_id` and `client_secret` generation.
- Empty `client_secret` when `client_type` is equal to `public`.

Programmatically

You can create a Client programmatically with Django shell `python manage.py shell`:

```
>>> from oidc_provider.models import Client
>>> c = Client(name='Some Client', client_id='123', client_secret='456', response_
↪type='code', redirect_uris=['http://example.com/'])
>>> c.save()
```

Read more about client creation from OAuth2 spec

CHAPTER 3

Server Keys

Server RSA keys are used to sign/encrypt ID Tokens. These keys are stored in the `RSAKey` model. So the package will automatically generate public keys and expose them in the `jwks_uri` endpoint.

You can easily create them with the admin:

Or by using `python manage.py creatersakey` command.

Here is an example response from the `jwks_uri` endpoint:

```
GET /openid/jwks HTTP/1.1
Host: localhost:8000

{
  "keys":[
    {
      "use":"sig",
      "e":"AQAB",
      "kty":"RSA",
      "alg":"RS256",
      "n":"3Gm0pS7ij_
↪SnY96wkbaki74MUYJrobXecO6xJhvmAEEhMHGpO0m4H2nbOWTf6Jc1FiiSvgvhObVk9xPOM6qMTQ5D5pfWZjNk99qDJXvAE4ImM
↪",
      "kid":"a38ea7fbf944cc060eaf5acc1956b0e3"
    }
  ]
}
```

CHAPTER 4

Templates

Add your own templates files inside a folder named `templates/oidc_provider/`. You can copy the sample html here and edit them with your own styles.

authorize.html:

```
<h1>Request for Permission</h1>

<p>Client <strong>{{ client.name }}</strong> would like to access this information of␣
↪you ...</p>

<form method="post" action="{% url 'oidc_provider:authorize' %}">

    {% csrf_token %}

    {{ hidden_inputs }}

    <ul>
    {% for scope in params.scope %}
        <li><strong>{{ scope.name }}</strong><br><i>{{ scope.description }}</i></li>
    {% endfor %}
    </ul>

    <input type="submit" value="Decline" />
    <input name="allow" type="submit" value="Authorize" />

</form>
```

error.html:

```
<h3>{{ error }}</h3>
<p>{{ description }}</p>
```

You can also customize paths to your custom templates by putting them in `OIDC_TEMPLATES` in the settings.

CHAPTER 5

Scopes and Claims

This subset of OpenID Connect defines a set of standard Claims. They are returned in the UserInfo Response.

The package comes with a setting called `OIDC_USERINFO`, basically it refers to a function that will be called with `claims` (dict) and `user` (user instance). It returns the `claims` dict with all the claims populated.

List of all the `claims` keys grouped by scopes:

profile	email	phone	address
name	email	phone_number	formatted
given_name	email_verified	phone_number_verified	street_address
family_name			locality
middle_name			region
nickname			postal_code
preferred_username			country
profile			
picture			
website			
gender			
birthdate			
zoneinfo			
locale			
updated_at			

How to populate standard claims

Somewhere in your Django `settings.py`:

```
OIDC_USERINFO = 'myproject.oidc_provider_settings.userinfo'
```

Then inside your `oidc_provider_settings.py` file create the function for the `OIDC_USERINFO` setting:

```
def userinfo(claims, user):
    # Populate claims dict.
    claims['name'] = '{0} {1}'.format(user.first_name, user.last_name)
    claims['given_name'] = user.first_name
    claims['family_name'] = user.last_name
    claims['email'] = user.email
    claims['address']['street_address'] = '...'

    return claims
```

Now test an Authorization Request using these scopes `openid profile email` and see how user attributes are returned.

Note: Please **DO NOT** add extra keys or delete the existing ones in the `claims` dict. If you want to add extra claims to some scopes you can use the `OIDC_EXTRA_SCOPE_CLAIMS` setting.

How to add custom scopes and claims

The `OIDC_EXTRA_SCOPE_CLAIMS` setting is used to add extra scopes specific for your app. Is just a class that inherit from `oidc_provider.lib.claims.ScopeClaims`. You can create or modify scopes by adding this methods into it:

- `info_scopename` class property for setting the verbose name and description.
- `scope_scopename` method for returning some information related.

Let's say that you want add your custom `foo` scope for your OAuth2/OpenID provider. So when a client (RP) makes an Authorization Request containing `foo` in the list of scopes, it will be listed in the consent page (`templates/oidc_provider/authorize.html`) and then some specific claims like `bar` will be returned from the `/userinfo` response.

Somewhere in your Django `settings.py`:

```
OIDC_EXTRA_SCOPE_CLAIMS = 'yourproject.oidc_provider_settings.CustomScopeClaims'
```

Inside your oidc_provider_settings.py file add the following class:

```
from django.utils.translation import ugettext as _
from oidc_provider.lib.claims import ScopeClaims

class CustomScopeClaims(ScopeClaims):

    info_foo = (
        _(u'Foo'),
        _(u'Some description for the scope.'),
    )

    def scope_foo(self):
        # self.user - Django user instance.
        # self.userinfo - Dict returned by OIDC_USERINFO function.
        # self.scopes - List of scopes requested.
        # self.client - Client requesting this claims.
        dic = {
            'bar': 'Something dynamic here',
        }
```

```
        return dic

    # If you want to change the description of the profile scope, you can redefine it.
    info_profile = (
        _(u'Profile'),
        _(u'Another description.'),
    )
```

Note: If a field is empty or `None` inside the dictionary you return on the `scope_scopename` method, it will be cleaned from the response.

CHAPTER 6

User Consent

The package store some information after the user grant access to some client. For example, you can use the `UserConsent` model to list applications that the user have authorized access. Like Google does here.

```
>>> from oidc_provider.models import UserConsent
>>> UserConsent.objects.filter(user__email='some@email.com')
[<UserConsent: Example Client - some@email.com>]
```

Properties

- `user`: Django user object.
- `client`: Relying Party object.
- `expires_at`: Expiration date of the consent.
- `scope`: Scopes authorized.
- `date_given`: Date of the authorization.

CHAPTER 7

OAuth2 Server

Because OIDC is a layer on top of the OAuth 2.0 protocol, this package gives you a simple but effective OAuth2 server that you can use not only for logging in your users on multiple platforms, also to protect some resources you want to expose.

Protecting Views

Here we are going to protect a view with a scope called `testscope`:

```
from django.http import JsonResponse
from django.views.decorators.http import require_http_methods

from oidc_provider.lib.utils.oauth2 import protected_resource_view


@require_http_methods(['GET'])
@protected_resource_view(['testscope'])
def protected_api(request, *args, **kwargs):

    dic = {
        'protected': 'information',
    }

    return JsonResponse(dic, status=200)
```

CHAPTER 8

Access Tokens

At the end of the login process, an access token is generated. This access token is the thing that's passed along with every API call (e.g. userinfo endpoint) as proof that the call was made by a specific person from a specific app.

Access tokens generally have a lifetime of only a couple of hours, you can use `OIDC_TOKEN_EXPIRE` to set custom expiration that suit your needs.

Obtaining an Access token

Go to the admin site and create a confidential client with `response_type = code` and `redirect_uri = http://example.org/`.

Open your browser and accept consent at:

```
http://localhost:8000/authorize?client_id=651462&redirect_uri=http://example.org/&
↪response_type=code&scope=openid email profile&state=123123
```

In the redirected URL you should have a `code` parameter included as query string:

```
http://example.org/?code=b9cedb346ee04f15ab1d3ac13da92002&state=123123
```

We use `code` value to obtain `access_token` and `refresh_token`:

```
curl -X POST \
    -H "Cache-Control: no-cache" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    "http://localhost:8000/token/" \
    -d "client_id=651462" \
    -d "client_secret=37b1c4ff826f8d78bd45e25bad75a2c0" \
    -d "code=b9cedb346ee04f15ab1d3ac13da92002" \
    -d "redirect_uri=http://example.org/" \
    -d "grant_type=authorization_code"
```

Example response:

```
{
    "access_token": "82b35f3d810f4cf49dd7a52d4b22a594",
    "token_type": "bearer",
    "expires_in": 3600,
    "refresh_token": "0bac2d80d75d46658b0b31d3778039bb",
    "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6..."
}
```

Then you can grab the access token and ask user data by doing a GET request to the `/userinfo` endpoint:

```
curl -X GET \
    -H "Cache-Control: no-cache" \
    "http://localhost:8000/userinfo/?access_token=82b35f3d810f4cf49dd7a52d4b22a594"
```

Expiration and Refresh of Access Tokens

If you receive a `401 Unauthorized` status when issuing access token probably means that has expired.

The RP application obtains a new access token by sending a POST request to the `/token` endpoint with the following request parameters:

```
curl -X POST \
    -H "Cache-Control: no-cache" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    "http://localhost:8000/token/" \
    -d "client_id=651462" \
    -d "client_secret=37b1c4ff826f8d78bd45e25bad75a2c0" \
    -d "grant_type=refresh_token" \
    -d "refresh_token=0bac2d80d75d46658b0b31d3778039bb"
```

CHAPTER 9

Session Management

The OpenID Connect Session Management 1.0 specification complements the core specification by defining how to monitor the End-User's login status at the OpenID Provider on an ongoing basis so that the Relying Party can log out an End-User who has logged out of the OpenID Provider.

Setup

Somewhere in your Django `settings.py`:

```
MIDDLEWARE_CLASSES = [
    ...
    'oidc_provider.middleware.SessionManagementMiddleware',
]

OIDC_SESSION_MANAGEMENT_ENABLE = True
```

If you're in a multi-server setup, you might also want to add `OIDC_UNAUTHENTICATED_SESSION_MANAGEMENT_KEY` to your settings and set it to some random but fixed string. While authenticated clients have a session that can be used to calculate the browser state, there is no such thing for unauthenticated clients. Hence this value. By default a value is generated randomly on startup, so this will be different on each server. To get a consistent value across all servers you should set this yourself.

Example RP iframe

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>RP Iframe</title>
</head>
<body onload="javascript:startChecking()">
```

```
    <iframe id="op-iframe" src="http://localhost:8000/check-session-iframe/"␣
↪frameborder="0" width="0" height="0"></iframe>
</body>
<script>
    var targetOP = "http://localhost:8000";

    window.addEventListener("message", receiveMessage, false);

    function startChecking() {
              checkStatus();
              setInterval('checkStatus()', 1000*60); // every 60 seconds
    }

    function checkStatus() {
              var clientId = '';
              var sessionState = '';
              var data = clientId + ' ' + sessionState;
              document.getElementById('op-iframe').contentWindow.postMessage(data,␣
↪targetOP);
    }

    function receiveMessage(event) {
        if (event.origin !== targetOP) {
            // Origin did not come from the OP.
            return;
        }
        if (event.data === 'unchanged') {
            // User is still logged in to the OP.
        } else if (event.data === 'changed') {
            // Perform re-authentication with prompt=none to obtain the current␣
↪session state at the OP.
        } else {
            // Error.
            console.log('Something goes wrong!');
        }
    }
</script>
</html>
```

RP-Initiated Logout

An RP can notify the OP that the End-User has logged out of the site, and might want to log out of the OP as well. In this case, the RP, after having logged the End-User out of the RP, redirects the End-User's User Agent to the OP's logout endpoint URL.

This URL is normally obtained via the `end_session_endpoint` element of the OP's Discovery response.

Parameters that are passed as query parameters in the logout request:

- **`id_token_hint`** Previously issued ID Token passed to the logout endpoint as a hint about the End-User's current authenticated session with the Client.
- **`post_logout_redirect_uri`** URL to which the RP is requesting that the End-User's User Agent be redirected after a logout has been performed.
- **`state`** OPTIONAL. Opaque value used by the RP to maintain state between the logout request and the callback to the endpoint specified by the `post_logout_redirect_uri` query parameter.

Example redirect:

```
http://localhost:8000/end-session/?id_token_hint=eyJhbGciOiJSUzI1NiIsImtpZCI6ImQwM...&
↪post_logout_redirect_uri=http://rp.example.com/logged-out/&state=c91c03ea6c46a86
```

CHAPTER 10

Settings

Customize your provider so fit your project needs.

OIDC_LOGIN_URL

OPTIONAL. `str`. Used to log the user in. By default Django's `LOGIN_URL` will be used. Read more in Django docs

`str`. Default is `/accounts/login/` (Django's `LOGIN_URL`).

SITE_URL

OPTIONAL. `str`. The OP server url.

If not specified will be automatically generated using `request.scheme` and `request.get_host()`.

For example `http://localhost:8000`.

OIDC_AFTER_USERLOGIN_HOOK

OPTIONAL. `str`. A string with the location of your function. Provide a way to plug into the process after the user has logged in, typically to perform some business logic.

Default is:

```
def default_hook_func(request, user, client):
    return None
```

Return `None` if you want to continue with the flow.

The typical situation will be checking some state of the user or maybe redirect him somewhere. With request you have access to all OIDC parameters. Remember that if you redirect the user to another place then you need to take him back to the authorize endpoint (use `request.get_full_path()` as the value for a "next" parameter).

OIDC_AFTER_END_SESSION_HOOK

OPTIONAL. `str`. A string with the location of your function. Provide a way to plug into the log out process just before calling Django's log out function, typically to perform some business logic.

Default is:

```
def default_after_end_session_hook(request, id_token=None, post_logout_redirect_
↪uri=None, state=None, client=None, next_page=None):
    return None
```

Return `None` if you want to continue with the flow.

OIDC_CODE_EXPIRE

OPTIONAL. `int`. Code object expiration after been delivered.

Expressed in seconds. Default is `60*10`.

OIDC_EXTRA_SCOPE_CLAIMS

OPTIONAL. `str`. A string with the location of your class. Default is `oidc_provider.lib.claims.ScopeClaims`.

Used to add extra scopes specific for your app. OpenID Connect RP's will use scope values to specify what access privileges are being requested for Access Tokens.

Read more about how to implement it in *Scopes and Claims* section.

OIDC_IDTOKEN_EXPIRE

OPTIONAL. `int`. ID Token expiration after been delivered.

Expressed in seconds. Default is `60*10`.

OIDC_IDTOKEN_PROCESSING_HOOK

OPTIONAL. `str` or `(list, tuple)`.

A string with the location of your function hook or `list` or `tuple` with hook functions. Here you can add extra dictionary values specific for your app into id_token.

The `list` or `tuple` is useful when you want to set multiple hooks, i.e. one for permissions and second for some special field.

The function receives a `id_token` dictionary and `user` instance and returns it with additional fields.

Default is:

```
def default_idtoken_processing_hook(id_token, user):

    return id_token
```

OIDC_IDTOKEN_SUB_GENERATOR

OPTIONAL. `str`. A string with the location of your function. `sub` is a locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client.

The function receives a `user` object and returns a unique `string` for the given user.

Default is:

```
def default_sub_generator(user):

    return str(user.id)
```

OIDC_SESSION_MANAGEMENT_ENABLE

OPTIONAL. `bool`. Enables OpenID Connect Session Management 1.0 in your provider. Read *Session Management* section.

Default is `False`.

OIDC_UNAUTHENTICATED_SESSION_MANAGEMENT_KEY

OPTIONAL. Supply a fixed string to use as browser-state key for unauthenticated clients. Read *Session Management* section.

Default is a string generated at startup.

OIDC_SKIP_CONSENT_EXPIRE

OPTIONAL. `int`. User consent expiration after been granted.

Expressed in days. Default is `30*3`.

OIDC_TOKEN_EXPIRE

OPTIONAL. `int`. Token object (access token) expiration after been created.

Expressed in seconds. Default is `60*60`.

OIDC_USERINFO

OPTIONAL. `str`. A string with the location of your function. Read *Scopes and Claims* section.

The function receives a `claims` dictionary with all the standard claims and `user` instance. Must returns the `claims` dict again.

Example usage:

```
def userinfo(claims, user):

    claims['name'] = '{0} {1}'.format(user.first_name, user.last_name)
    claims['given_name'] = user.first_name
    claims['family_name'] = user.last_name
    claims['email'] = user.email
    claims['address']['street_address'] = '...'

    return claims
```

Note: Please **DO NOT** add extra keys or delete the existing ones in the `claims` dict. If you want to add extra claims to some scopes you can use the `OIDC_EXTRA_SCOPE_CLAIMS` setting.

OIDC_GRANT_TYPE_PASSWORD_ENABLE

OPTIONAL. A boolean to set whether to allow the Resource Owner Password Credentials Grant. https://tools.ietf.org/html/rfc6749#section-4.3

Important: From the specification: "Since this access token request utilizes the resource owner's password, the authorization server **MUST** protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts)."

There are many ways to implement brute force attack prevention. We cannot decide what works best for you, so you will have to implement a solution for this that suits your needs.

OIDC_TEMPLATES

OPTIONAL. A dictionary pointing to templates for authorize and error pages. Default is:

```
{
    'authorize': 'oidc_provider/authorize.html',
    'error': 'oidc_provider/error.html'
}
```

The following contexts will be passed to the `authorize` and `error` templates respectively:

```
# For authorize template
{
    'client': 'an instance of Client for the auth request',
    'hidden_inputs': 'a rendered html with all the hidden inputs needed for␣
↪AuthorizeEndpoint',
```

```
    'params': 'a dict containing the params in the auth request',
    'scopes': 'a list of scopes'
}

# For error template
{
    'error': 'string stating the error',
    'description': 'string stating description of the error'
}
```

Note: The templates that are not specified here will use the default ones.

CHAPTER 11

Signals

Use signals in your application to get notified when some actions occur.

For example:

```
from django.dispatch import receiver

from oidc_provider.signals import user_decline_consent


@receiver(user_decline_consent)
def my_callback(sender, **kwargs):
    print(kwargs)
    print('Ups! Some user has declined the consent.')
```

user_accept_consent

Sent when a user accept the authorization page for some client.

user_decline_consent

Sent when a user decline the authorization page for some client.

CHAPTER 12

Examples

Pure JS client using Implicit Flow

Testing OpenID Connect flow can be as simple as putting one file with a few functions on the client and calling the provider. Let me show.

01. Setup the provider

You can use the example project code to run your OIDC Provider at `localhost:8000`.

Go to the admin site and create a public client with a response_type `id_token token` and a redirect_uri `http://localhost:3000`.

Note: Remember to create at least one **RSA Key** for the server. `python manage.py creatersakey`

02. Create the client

As relying party we are going to use a JS library created by Nat Sakimura. Here is the article.

index.html:

```
<!DOCTYPE html>
<html>
<head>

    <title>OIDC RP</title>

</head>
<body>

    <center>
        <h1>OpenID Connect RP Example</h1>
        <button id="login-button">Login</button>
    </center>
```

```
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.2.2/jquery.min.js"></
↪script>
    <script src="https://www.sakimura.org/test/openidconnect.js"></script>

    <script type="text/javascript">
    $(function() {
        var clientInfo = {
            client_id : '',
            redirect_uri : 'http://localhost:3000'
        };

        OIDC.setClientInfo(clientInfo);

        var providerInfo = OIDC.discover('http://localhost:8000');

        OIDC.setProviderInfo(providerInfo);
        OIDC.storeInfo(providerInfo, clientInfo);

        // Restore configuration information.
        OIDC.restoreInfo();

        // Get Access Token
        var token = OIDC.getAccessToken();

        // Make userinfo request using access_token.
        if (token !== null) {
            $.get('http://localhost:8000/userinfo/?access_token='+token, function(␣
↪data ) {
                alert('USERINFO: '+ JSON.stringify(data));
            });
        }

        // Make an authorization request if the user click the login button.
        $('#login-button').click(function (event) {
            OIDC.login({
                scope : 'openid profile email',
                response_type : 'id_token token'
            });
        });
    });
    </script>

</body>
</html>
```

Note: Remember that you must set your client_id (line 21).

03. Make an authorization request

By clicking the login button an authorization request has been made to the provider. After you accept it, the provider will redirect back to your previously registered `redirect_uri` with all the tokens requested.

04. Requesting user information

Now having the access_token in your hands you can request the user information by making a request to the `/userinfo` endpoint of the provider.

In this example we display information in the alert box.

CHAPTER 13

Contribute

We love contributions, so please feel free to fix bugs, improve things, provide documentation. You SHOULD follow this steps:

- Fork the project.
- Make your feature addition or bug fix.
- Add tests for it inside `oidc_provider/tests`. Then run all and ensure everything is OK (read docs for how to test in all envs).
- Send pull request to the specific version branch.

Running Tests

Use tox for running tests in each of the environments, also to run coverage among:

```
# Run all tests.
$ tox

# Run with Python 2.7 and Django 1.9.
$ tox -e py27-django19

# Run single test file.
$ python runtests.py oidc_provider.tests.test_authorize_endpoint
```

Also tests run on every commit to the project, we use travis for this.

Improve Documentation

We use Sphinx for generate this documentation. I you want to add or modify something just:

- Install Sphinx (`pip install sphinx`) and the auto-build tool (`pip install sphinx-autobuild`).

- Move inside the docs folder. `cd docs/`
- Generate and watch docs by running `sphinx-autobuild . _build/`.
- Open `http://127.0.0.1:8000` on a browser.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search