invenio-oauth2server Documentation

Release 1.0.1

CERN

May 25, 2018

Contents

Invenio module that implements OAuth 2 server.

- Free software: MIT license
- Documentation: https://invenio-oauth2server.readthedocs.io/

CHAPTER 1

Features

- **Implements the OAuth 2.0 authentication protocol.**
  - Provides REST API to provide access tokens.
  - Provides decorators that can be used to restrict access to resources.
- Handles authentication using JSON Web Tokens.
- Adds support for CSRF protection in REST API.

1.1 User's Guide

This part of the documentation will show you how to get started in using Invenio-OAuth2Server.

1.1.1 Overview

Understanding API Authentication in Invenio

A user can make authenticated requests against the Invenio REST APIs using two different methods:

Session

A user who logged into an Invenio application in the browser obtains a session. The session is implemented via Secure HTTP-only cookies, to ensure that the cookie containing the session identifier is only submitted over HTTPS, and that JavaScript applications running in the browser cannot access the cookie. When the session cookie is provided in an HTTP request to the API, the cookie is used to authenticate the user.

Because the session-based authentication is primarily used from a browser, it is important to protect the API against Cross-Site Request Forgery (CSRF) attacks. Invenio protects against CSRF-attacks by embedding a short lived CSRF-token into the HTML DOM tree from the server-side. This CSRF-token is then read by a JavaScript application and added to the HTTP request header. Thus, the HTTP request header will include both the session cookie as well as the CRSF-token. The CSRF-token is implemented via a JSON Web Token (JWT).

The session identifier stored inside the session cookie is furthermore protected in a way, so that it must be used from the same machine and same browser.

Access token

An access token (or API key) can also be used to make authenticated requests to the API. Access tokens are primarily used by machines accessing the Invenio REST API, contrary to session-based authentication which is primarily used in browsers by humans. The access tokens can also be used to delegate user rights to a third-party application without exposing user credentials. This delegation of rights can further be scoped to specific parts of the API, to not give full access to third-party applications.

Access tokens can be obtained in different ways. A user may for instance manually create an access token via the user interface, or e.g. a third-party application can initiate an OAuth 2.0 authentication flow that eventually provides them with an access token. The different scenarios for how to obtain an access token are explained in detail further below.

Obtaining a session and JWT token

To obtain a session:

1. The user logs in by providing his login credentials
2. A new session is created

After this point we can add a CSRF token to not be prone to CSRF attacks.

3. For the CSRF token we can use the JWT as it contains user information and it fulfills the key properties of a CSRF token

By default, the JWT is embedded in the DOM tree using the Jinja context processor `{{jwt()}}` or `{{jwt_token()}}` from a template. By passing the JWT with each request, the user state is never saved in server memory making this a stateless authentication mechanism. Then the server just looks for and validates the JWT in the `Authorization` header, to allow access to the protected resources.

Obtaining an access token

In the case where the client requesting an access token is the resource owner, the token will allow all permissions the user would have by providing his username and password credentials. For example to use a personal access token to send REST API requests, the procedure is the following:

1. First the user logs in and navigates to his profile page
2. Clicks on `Create New Personal Token`
3. Stores the generated string in a variable `$ACCESS_TOKEN`
4. Now requests can be made to protected resources by passing it as a parameter, e.g.

```
curl -XPOST -d '{some_record_data}' $HOST:5000/records/
\?access_token=$ACCESS_TOKEN
```

In the case where the client is a third party, a web application for example requesting access to an owner's protected resources, the procedure is different. Let's see the case where a user goes to `example.com` and chooses to log in via Invenio. The setup to enable this is as follows:

1. First, the application has to be registered as an authorized application
2. This can be done by the settings page, as it was for the personal access tokens, but now clicking on `New Application`
3. After filling out the form and setting a `Redirect URL`, a `Client ID` and `Client Secret` are generated

4. These have to be set in the `example.com` application, in order to be able to make requests

Now a user can navigate to `example.com` and can select to log in via Invenio. The procedure will be along the following lines:

5. A request is sent to the `/authorize` endpoints in Invenio from `example.com` with the `Client ID` and `Client Secret` passed as parameters
6. The user is redirected to an Invenio page where he is asked to log in
7. After logging in, a form shows what type of permissions the `example.com` is requesting, and the user can decide to authorize it
8. Invenio redirects back to the `Redirect URL` set for this application, and returns an authorization code with a limited lifespan
9. The authorization can be used to obtain an access token by querying the `/token` endpoint in Invenio
10. `example.com` can now send requests to Invenio using this access token to access resources available to the user

Oauth2 flows

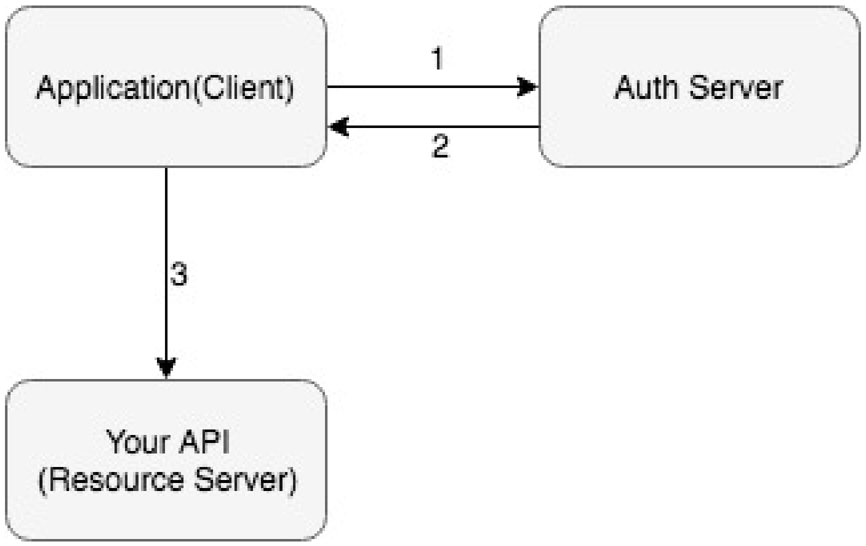
There are different Oauth2 flows you should use depending mostly on the type of your `Client` but also in other parameters such as the level of trust of the `Client`. By different flows we mean that Oauth2 provides different grant types that you can use. `Grant types` are different ways of retrieving an `access token` that eventually will lead you to access a protected resource. Before analyzing the different Oauth2 flows let's see some Oauth2 terminology:

- **Resource owner**: the entity that has the ownership of a protected resource. Can be an the application itself or an end user.
- **Client**: an application that requests access to a protected resource on behalf of the resource owner.
- **Resource server**: the server in which the protected resource is stored. This is the API you want to have access.
- **Authorization server**: this is the server that authenticates the resource owner and issues an access token to the Client after getting proper authorization. In our case this is the OAuth2Server package.
- **User Agent**: the agent used by the Resource Owner to interact with the Client, for example a browser or a native application.

The crucial thing to decide which Oauth2 grant type is most suitable for you to use, as we said, is the type of your client. Having in mind that we define the below 4 cases.

Client is the resource owner

This is the case that the application that requests access to a protected resource is also the owner of this resource. In that case the application holds the `Client ID` and the `Client Secret` and uses them to authenticate itself through the authentication server and retrieve the access token. Such an example could be a service running on the client server and trying to get access to a resource on the same server. A typical flow diagram is the following:

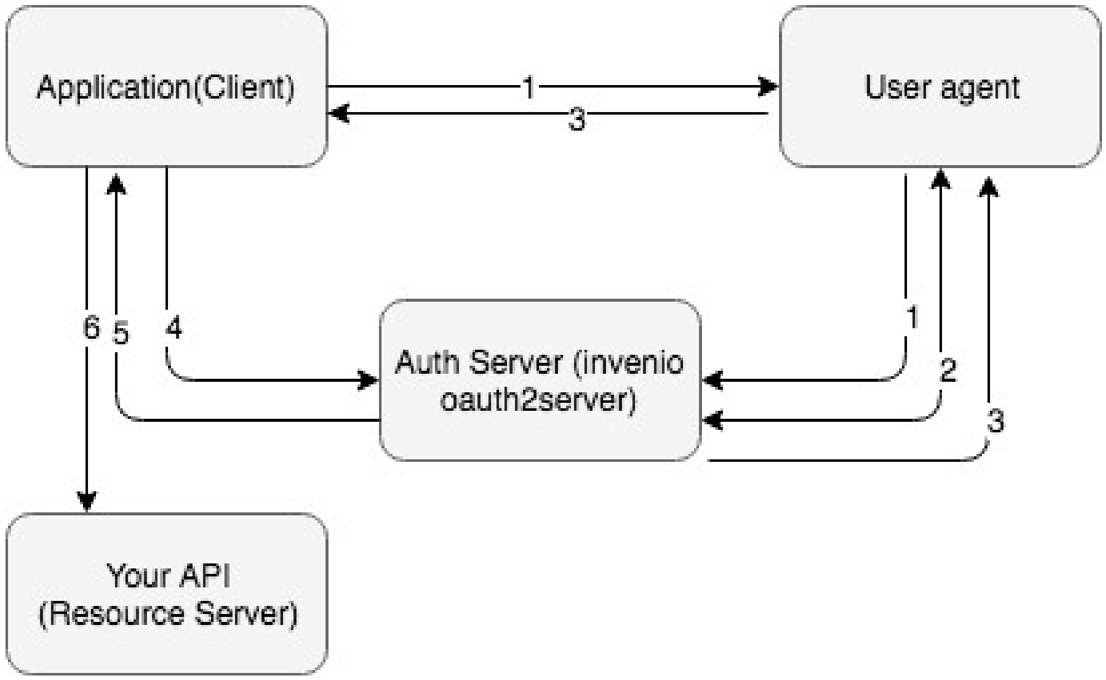


1. Application authenticates itself using Client ID and Secret.
2. Retrieves an access token.
3. Uses the token to access the protected resource.

If this case is the one that suits your needs then you should use the `Client Credentials grant`.

Client is an application running on a web server

In that case you should use the `Authorization Code grant`. In this flow the Client requests an access token from the authorization server in order to access the protected resource. The Client gets an access token, and optionally a refresh token, after first the resource owner is authorized.

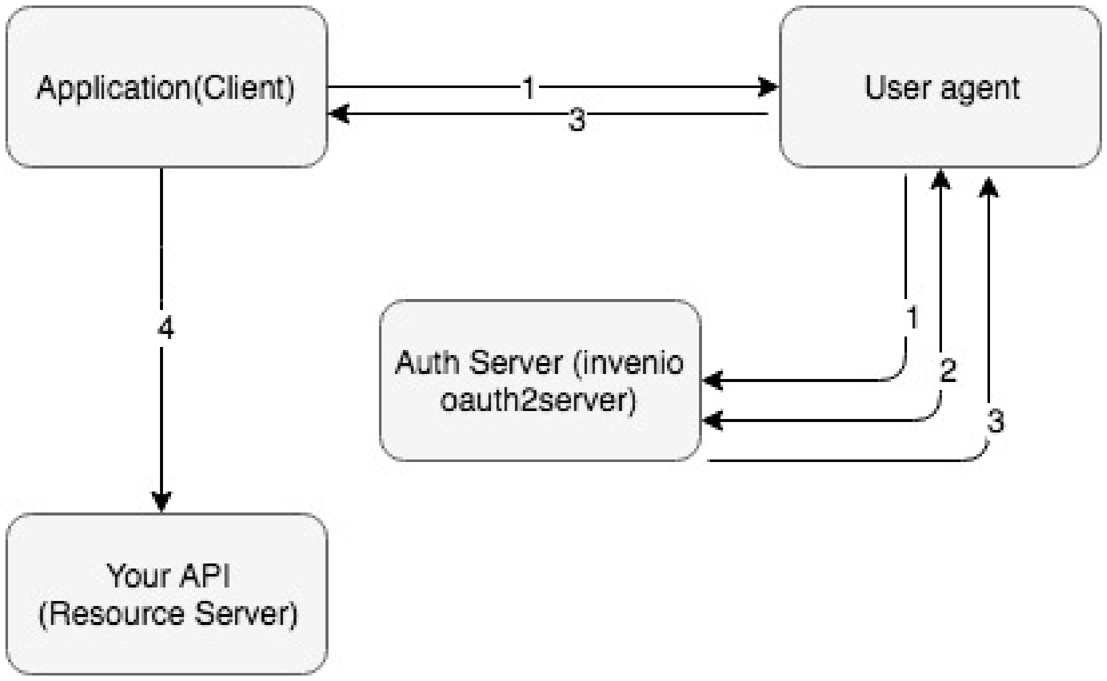


1. Application redirects the user agent to the /authorize url to authenticate itself through the authorization server.

2. The end user the first time is provided with a consent page that asks for specific permissions to be granted to the application (e.g. user email, list of contacts etc.)
3. After the user confirms the access grant the authorization server returns an authorization code to the application.
4. With the possession of the authorization code, the application asks from the authorization server an access token in exchange for its code.
5. The authorization server validates the code sent from the application and if is valid issues an access token back to it. Optionally can return also a refresh token that is used by the application when the access token is expired.
6. The application uses the retrieved access token to eventually consume the protected resources stored in the resource server.

Client is a Single Page Application

If your application is a single page application then you should use the `Implicit grant`. In this grant type instead of getting first an authorization code in order to ask for an access token you directly ask for the token. In the plus side this method is faster as there is no need for round trip to get an access token. However, there is a security risk as the access token is exposed to the user agent (e.g. the user's browser). Also you should consider that the `Implicit grant` doesn't return refresh tokens.


Application(Client)
User agent
Auth Server (invenio oauth2server)
Your API (Resource Server)
1
3
4
1
2
3

1. Application redirects the user agent to the /authorize url to authenticate itself through the authorization server.
2. The end user the first time is provided with a consent page that asks for specific permissions to be granted to the application (e.g. user email, list of contacts etc.)
3. After the user confirms the access grant the authorization server returns an access token to the application. Note that in this flow no refresh token is issued and the access_token is short lived.
4. The application uses the retrieved access token to eventually consume the protected resources stored in the resource server.

Client is trusted with user Credentials

In that case probably you should use the `Resource Owner Password Credentials Grant`. In this flow the end user trusts the `Client` with his/her credentials in order to be used by the client to authenticate him/her through the authorization server. This grant type is disabled by default in Invenio-OAuth2Server, and should only be used if there is no possibily to use another redirect-based flow.

1.1.2 Installation

Invenio-OAuth2Server is on PyPI so all you need is:

```
$ pip install invenio-oauth2server
```

1.1.3 Configuration

OAuth2Server configuration variables.

`invenio_oauth2server.config.`**`OAUTH2SERVER_ALLOWED_GRANT_TYPES = set(['client_credentials',`**

A set of allowed grant types.

The allowed values are `authorization_code`, `password`, `client_credentials`, `refresh_token`). By default password is disabled, as it requires the client application to gain access to the username and password of the resource owner.

`invenio_oauth2server.config.`**`OAUTH2SERVER_ALLOWED_RESPONSE_TYPES = set(['token', 'code'])`**

A set of allowed response types.

The allowed values are `code` and `token`.

- `code` is used for authorization_code grant types
- `token` is used for implicit grant types

`invenio_oauth2server.config.`**`OAUTH2SERVER_ALLOWED_URLENCODE_CHARACTERS = '=&;:%+~,*@!()/?'`**

A string of special characters that should be valid inside a query string.

See also:

See *`monkeypatch_oauthlib_urlencode_chars`* for a full explanation.

`invenio_oauth2server.config.`**`OAUTH2SERVER_CLIENT_ID_SALT_LEN = 40`**

Length of client id.

`invenio_oauth2server.config.`**`OAUTH2SERVER_CLIENT_SECRET_SALT_LEN = 60`**

Length of the client secret.

`invenio_oauth2server.config.`**`OAUTH2SERVER_JWT_AUTH_HEADER = 'Authorization'`**

Header for the JWT.

Note: Authorization: Bearer xxx

`invenio_oauth2server.config.`**`OAUTH2SERVER_JWT_AUTH_HEADER_TYPE = 'Bearer'`**

Header Authorization type.

Note: By default the authorization type is `Bearer` as recommented by JWT invenio_oauth2server.config.**OAUTH2SERVER_JWT_VERYFICATION_FACTORY = 'invenio_oauth2server.u**

Import path of factory used to verify JWT.

The `request.headers` should be passed as parameter.

invenio_oauth2server.config.**OAUTH2SERVER_TOKEN_PERSONAL_SALT_LEN = 60**

Length of the personal access token.

invenio_oauth2server.config.**OAUTH2_CACHE_TYPE = 'redis'**

Type of cache to use for storing the temporary grant token.

invenio_oauth2server.config.**OAUTH2_PROVIDER_ERROR_ENDPOINT = 'invenio_oauth2server.errors'**

Error view endpoint.

invenio_oauth2server.config.**OAUTH2_PROVIDER_TOKEN_EXPIRES_IN = 3600**

Life time of an access token.

1.1.4 Usage

Invenio module that implements OAuth 2 server.

Protecting your REST API with authentication

If you want to have your REST API endpoints protected using OAuth you should register their blueprint inside the API app (InvenioOAuth2ServerREST) which by default includes a `before_request` hook. This hook will, if there is an OAuth token, verify it and set the current user accordingly. It is important to highlight that this configuration allows either authenticated clients or anonymous clients.

In case you need to allow access to a resource only for authenticated clients, you should use the require_api_auth decorator which requires OAuth2 login:

```
@app.route('/api/resource', methods=['GET'])
@require_api_auth()
def index():
    return 'Protected resource'
```

However, protecting your resources only with authentication is not recommended. Instead, you should add an extra layer using always scopes. This is because, basically, any client that owns a token has control over every user resource. Therefore, using scopes gives a fine-grain control. Here an example using the default `email_scope`:

```
from invenio_oauth2server.scopes import email_scope
from flask_login import current_user

@app.route('/api/email', methods=['GET'])
@require_api_auth()
@require_oauth_scopes(email_scope.id_)
def index():
    return current_user.email
```

Delegating rights via scopes

As mentioned before, the recommended way to protect your endpoints is to use fine-grain control with scopes. Invenio-OAuth2Server offers the possibility to create new ones:

```python
from invenio_oauth2server.models import Scope

homepage_read = Scope('homepage:read',
                      help_text='Access to the homepage',
                      group='test')
```

Next, you should add them to `setup.py` entrypoints so they get initialized at start up:

```python
setup(
    ...
    entry_points={
        'invenio_oauth2server.scopes': [
            'homepage_read = path.to.scopes.file:homepage_read',
        ]
    }
    ...
)
```

And then, they can be used in your application:

```python
from path.to.scopes.file import homepage_read

@app.route('/', methods=['GET'])
@require_api_auth()
@require_oauth_scopes(homepage_read.id_)
def index():
    return 'Front page content.'
```

So, finally, with this example, we would allow any authenticated client with rights to use the `homepage_scope` to read the homepage but, prevent from reading the email if they do not have rights for using the `email_scope`.

To test this features you can build your own application or use the provided *example app* as boilerplate.

Access control

It is important to remember that the usage of authentication and scopes is not enough in most of the cases so access control need to be configured as well. For more information about access control in Invenio you can visit Invenio-Access documentation.

1.1.5 Example applications

Example application

Run example development server:

```
$ pip install -e .[all]
$ cd examples
$ ./app-setup.sh
$ ./app-fixtures.sh
$ FLASK_APP=app.py flask run -p 5000
```

Open settings page to generate a token:

```
$ open http://127.0.0.1:5000/account/settings/applications
```

Login with:

> username: admin@inveniosoftware.org
> password: 123456

Click on "New token" and compile the form: insert the name "foobar", check scope "test:scope" and click "create". The server will show you the generated Access Token.

Make a request to test the token:

```
export TOKEN=<generated Access Token>
curl -i -X GET -H "Content-Type:application/json" http://127.0.0.1:5000/ \
    -H "Authorization:Bearer $TOKEN"
```

To end and remove any traces of example application, stop the example application and run: .. code-block:: console

> $ ./app-teardown.sh

Example OAuth2 Consumer

This example OAuth2 consumer application is used to fetch an OAuth2 access token from example application.

For more information about OAuth2 protocol see
https://invenio-oauthclient.readthedocs.io/en/latest/overview.html

Note: Before continuing make sure example application is running.

Open settings page of example app to register a new OAuth2 application:

```
$ open http://127.0.0.1:5000/account/settings/applications
```

Login using:

> **username:** admin@inveniosoftware.org
> **password:** 123456

Click on "New application" and compile registration form with following data:

> **Name:** foobar-app
> **Description:** An example OAuth2 consumer application
> **Website URL:** http://127.0.0.1:5100/
> **Redirect URIs:** http://127.0.0.1:5100/authorized
> **Client Type:** Confidential

Click register and example application will generate and show you a Client ID and Client Secret.

Open another terminal and move to examples-folder.

Export these values using following environment variables before starting the example consumer or change values of corresponding keys in *examples/consumer.py* to match.

```
$ export CONSUMER_CLIENT_ID=<generated_client_id>
$ export CONSUMER_CLIENT_SECRET=<generated_client_secret>
```

LOGOUT admin@inveniosoftware.org from example application:

```
$ open http://127.0.0.1:5000/logout
```

Run the example consumer

```
$ FLASK_APP=consumer.py flask run -p 5100
```

Start OAuth authorization flow and you will be redirected to example application for authentication and to authorize example consumer to access your account details on example application.

Login to example application with:

> **username:** reader@inveniosoftware.org
> **password:** 123456

Review the authorization request presented to you and authorize the example consumer.

You will be redirected back to example consumer where you can see details of the authorization token that example application generated to example consumer.

Note: In case the authorization flow ends in an error, you can usually see the error in query-part of the URL.

Using example consumer's UI you can request a new access token from example application either by using a refresh token or by completing the authorization flow again.

To manage settings of OAuth2 consumer at invenio-oauth2server settings page, login with the account that registered the consumer, admin@inveniosoftware.org.

To review and possibly revoke permissions of OAuth2 consumer that has been authorized to access resources login with the account that authorized the consumer, reader@inveniosoftware.org.

This example consumer is inspired by example presented in requests-oauthlib documentation (http://requests-oauthlib.rtfd.io/en/latest/examples/real_world_example_with_refresh.html) and is based on example application(s) of flask-oauthlib: (https://github.com/lepture/flask-oauthlib/tree/master/example) (https://github.com/lepture/flask-oauthlib/tree/master/example/contrib/experiment-client/douban.py)

Note that to support automatic refreshing of access tokens this consumer uses flask-oauthlib.contrib.client which is considered experimental.

1.2 API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

1.2.1 API Docs

Invenio module that implements OAuth 2 server.

class `invenio_oauth2server.ext.`**`InvenioOAuth2Server`**(*app=None*, ***kwargs*)
Invenio-OAuth2Server extension.

Extension initialization.

Parameters **`app`** – An instance of `flask.Flask`.

`init_app`(*app*, *entry_point_group='invenio_oauth2server.scopes'*, ***kwargs*)
Flask application initialization.

Parameters

- **`app`** – An instance of `flask.Flask`.
- **`entry_point_group`** – The entrypoint group name to load plugins. (Default: `'invenio_oauth2server.scopes'`)

`init_config`(*app*)
Initialize configuration.

Parameters **`app`** – An instance of `flask.Flask`.

class `invenio_oauth2server.ext.`**`InvenioOAuth2ServerREST`**(*app=None*, ***kwargs*)
Invenio-OAuth2Server REST extension.

Extension initialization.

Parameters **`app`** – An instance of `flask.Flask`.

`init_app`(*app*, ***kwargs*)
Flask application initialization.

Parameters **`app`** – An instance of `flask.Flask`.

`init_config`(*app*)
Initialize configuration.

static **`monkeypatch_oauthlib_urlencode_chars`**(*chars*)
Monkeypatch OAuthlib set of "URL encoded"-safe characters.

Note: OAuthlib keeps a set of characters that it considers as valid inside an URL-encoded query-string during parsing of requests. The issue is that this set of characters wasn't designed to be configurable since it should technically follow various RFC specifications about URIs, like for example RFC3986. Many online services and frameworks though have designed their APIs in ways that aim at keeping things practical and readable to the API consumer, making use of special characters to mark or seperate query-string arguments. Such an example is the usage of embedded JSON strings inside query-string arguments, which of course have to contain the "colon" character (:) for key/value pair definitions.

Users of the OAuthlib library, in order to integrate with these services and frameworks, end up either circumventing these "static" restrictions of OAuthlib by pre-processing query-strings, or -in search of a more permanent solution- directly make Pull Requests to OAuthlib to include additional characters in the set, and explain the logic behind their decision (one can witness these efforts inside the git history of the source file that includes this set of characters here). This kind of tactic leads easily to misconceptions about the ability one has over the usage of specific features of services and frameworks. In order to tackle this issue in Invenio-OAuth2Server, we are monkey-patching this set of characters using a configuration variable, so that usage of any special characters is a conscious decision of the package user.

`invenio_oauth2server.ext.`**`verify_oauth_token_and_set_current_user`**`()`

Verify OAuth token and set current user on request stack.

This function should be used **only** on REST application.

```
app.before_request(verify_oauth_token_and_set_current_user)
```

Decorators

Useful decorators for checking authentication and scopes.

`invenio_oauth2server.decorators.`**`require_api_auth`**`(`*allow_anonymous=False*`)`

Decorator to require API authentication using OAuth token.

Parameters **`allow_anonymous`** – Allow access without OAuth token (default: `False`).

`invenio_oauth2server.decorators.`**`require_oauth_scopes`**`(`**scopes*`)`

Decorator to require a list of OAuth scopes.

Decorator must be preceded by a `require_api_auth()` decorator. Note, API key authentication is bypassing this check.

Parameters **`*scopes`** – List of scopes required.

Models

OAuth2Server models.

`class` `invenio_oauth2server.models.`**`Client`**`(`***kwargs*`)`

A client is the app which want to use the resource of a user.

It is suggested that the client is registered by a user on your site, but it is not required.

The client should contain at least these information:

client_id: A random string client_secret: A random string client_type: A string represents if it is confidential redirect_uris: A list of redirect uris default_redirect_uri: One of the redirect uris default_scopes: Default scopes of the client

But it could be better, if you implemented:

allowed_grant_types: A list of grant types allowed_response_types: A list of response types validate_scopes: A function to validate scopes

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

allowed_grant_types
Return allowed grant types.

allowed_response_types
Return allowed response types.

client_id
Client application ID.

client_secret
Client application secret.

client_type
Return client type.

default_redirect_uri
Return default redirect uri.

default_scopes
List of default scopes for client.

description
Human readable description.

gen_salt()
Generate salt.

get_users
Get number of users.

is_confidential
Determine if client application is public or not.

is_internal
Determins if client application is an internal application.

name
Human readable name of the application.

redirect_uris
Return redirect uris.

reset_client_id()
Reset client id.

reset_client_secret()
Reset client secret.

user
Relationship to user.

user_id
Creator of the client application.

validate_scopes(*scopes*)
Validate if client is allowed to access scopes.

class invenio_oauth2server.models.**NoneAesEngine**
Filter None values from encrypting.

decrypt(*value*)
Decrypt value on the way out.

encrypt(*value*)
Encrypt a value on the way in.

class invenio_oauth2server.models.**OAuthUserProxy**(*user*)
Proxy object to an Invenio User.

Initialize proxy object with user instance.

check_password(*password*)
Check user password.

classmethod get_current_user()
Return an instance of current user object.

id
Return user identifier.

class invenio_oauth2server.models.**Scope**(*id_*, *help_text=''*, *group=''*, *internal=False*)
OAuth scope definition.

Initialize scope values.

class invenio_oauth2server.models.**Token**(***kwargs*)
A bearer token is the final token that can be used by the client.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

client
SQLAlchemy relationship to client application.

client_id
Foreign key to client application.

classmethod create_personal(*name*, *user_id*, *scopes=None*, *is_internal=False*)
Create a personal access token.

A token that is bound to a specific user and which doesn't expire, i.e. similar to the concept of an API key.

Parameters

- **name** – Client name.
- **user_id** – User ID.
- **scopes** – The list of permitted scopes. (Default: `None`)
- **is_internal** – If `True` it's a internal access token. (Default: `False`)

Returns A new access token.

get_visible_scopes()
Get list of non-internal scopes for token.

Returns A list of scopes.

id
Object ID.

is_internal
Determines if token is an internally generated token.

is_personal
Personal accesss token.

scopes
Return all scopes.

Returns A list of scopes.

token_type
Token type - only bearer is supported at the moment.

user
SQLAlchemy relationship to user.

user_id
Foreign key to user.

`invenio_oauth2server.models.`**secret_key**`()`
Return secret key as bytearray.

Provider

Configuration of Flask-OAuthlib provider.

`invenio_oauth2server.provider.`**get_client**`(`*client_id*`)`
Load the client.

Needed for grant_type client_credentials.

Add support for OAuth client_credentials access type, with user inactivation support.

Parameters **client_id** – The client ID.

Returns The client instance or `None`.

`invenio_oauth2server.provider.`**get_token**`(`*access_token=None*, *refresh_token=None*`)`
Load an access token.

Add support for personal access tokens compared to flask-oauthlib. If the access token is `None`, it looks for the refresh token.

Parameters

- **access_token** – The access token. (Default: `None`)
- **refresh_token** – The refresh token. (Default: `None`)

Returns The token instance or `None`.

`invenio_oauth2server.provider.`**get_user**`(`*email*, *password*, **args*, ***kwargs*`)`
Get user for grant type password.

Needed for grant type 'password'. Note, grant type password is by default disabled.

Parameters

- **email** – User email.
- **password** – Password.

Returns The user instance or `None`.

`invenio_oauth2server.provider.`**save_token**`(`*token*, *request*, **args*, ***kwargs*`)`
Token persistence.

Parameters

- **token** – A dictionary with the token data.
- **request** – The request instance.

Returns A *invenio_oauth2server.models.Token* instance.

Validators

Validators for OAuth 2.0 redirect URIs and scopes.

class invenio_oauth2server.validators.**URLValidator**(*require_tld=True*, *message=None*)
URL validator.

invenio_oauth2server.validators.**validate_redirect_uri**(*value*)
Validate a redirect URI.

Redirect URIs must be a valid URL and use https unless the host is localhost for which http is accepted.

Parameters **value** – The redirect URI.

invenio_oauth2server.validators.**validate_scopes**(*value_list*)
Validate if each element in a list is a registered scope.

Parameters **value_list** – The list of scopes.

Raises ***invenio_oauth2server.errors.ScopeDoesNotExists*** – The exception is raised if a scope is not registered.

Returns `True` if it's successfully validated.

Proxies

Helper proxy to the state object.

invenio_oauth2server.proxies.**current_oauth2server = <LocalProxy unbound>**
Return current state of the OAuth2 server extension.

Errors

Errors raised by Invenio-OAuth2Server.

exception invenio_oauth2server.errors.**JWTDecodeError**(*errors=None*, ***kwargs*)
Exception raised when decoding is failed.

Initialize JWTExtendedException.

exception invenio_oauth2server.errors.**JWTExpiredToken**(*errors=None*, ***kwargs*)
Exception raised when JWT is expired.

Initialize JWTExtendedException.

exception invenio_oauth2server.errors.**JWTExtendedException**(*errors=None*, ***kwargs*)

Base exception for all JWT errors.

Initialize JWTExtendedException.

get_body(*environ=None*)
Get the request body.

get_errors()
Get errors.

Returns A list containing a dictionary representing the errors.

get_headers(*environ=None*)
Get a list of headers.

exception invenio_oauth2server.errors.**JWTInvalidHeaderError**(*errors=None*, ***kwargs*)
Exception raised when header argument is missing.

Initialize JWTExtendedException.

exception invenio_oauth2server.errors.**JWTInvalidIssuer**(*errors=None*, ***kwargs*)
Exception raised when the user is not valid.

Initialize JWTExtendedException.

exception invenio_oauth2server.errors.**JWTNoAuthorizationError**(*errors=None*, ***kwargs*)
Exception raised when permission denied.

Initialize JWTExtendedException.

exception invenio_oauth2server.errors.**OAuth2ServerError**
Base class for errors in oauth2server module.

exception invenio_oauth2server.errors.**ScopeDoesNotExists**(*scope*, **args*, ***kwargs*)
Scope is not registered it scopes registry.

Initialize exception by storing invalid scope.

1.3 Additional Notes

Notes on how to contribute, legal information and changes are here for the interested.

1.3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

Types of Contributions

Report Bugs

Report bugs at https://github.com/inveniosoftware/invenio-oauth2server/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Invenio-OAuth2Server could always use more documentation, whether as part of the official Invenio-OAuth2Server docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/inveniosoftware/invenio-oauth2server/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *invenio-oauth2server* for local development.

1. Fork the *inveniosoftware/invenio-oauth2server* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/invenio-oauth2server.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv invenio-oauth2server
$ cd invenio-oauth2server/
$ pip install -e .[all]
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

```
$ ./run-tests.sh
```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8 as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -s -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.com/inveniosoftware/invenio-oauth2server/pull_requests and make sure that the tests pass for all supported Python versions.

1.3.2 Changes

Version 1.0.1 (released 2018-05-25)

- Flask v1.0 support.

Version 1.0.0 (released 2018-03-23)

- Initial public release.

1.3.3 License

1.3.4 Contributors

- Alexander Ioannidis
- Alizee Pace
- David Caro
- Diego Rodriguez
- Dinos Kousidis
- Harri Hirvonsalo
- Harris Tzovanakis
- Jacopo Notarstefano
- Jiri Kuncar
- Krzysztof Nowak
- Lars Holm Nielsen
- Leonardo Rossi
- Miltiadis Alexis
- Nicolas Harraudeau
- Paulina Lach
- Sami Hiltunen
- Sebastian Witowski
- Tibor Simko

Python Module Index

Index