# flask-oauth2-devices Documentation

***Release 0.0.1***

**Joe Cabrera**

February 19, 2015

Contents

**Abstract**

Flask-oauth2-devices is an extension to Flask that helps you to create the device flow for OAuth2 providers. It is based on the implementation provided by Google and Section 3.7 of the OAuth 2.05 specification.

CHAPTER 1

# Features

- Support for OAuth2 device flow servers
- Friendly API (similar to flask-oauthlib)

## 1.1 Guide

### 1.1.1 Introduction

Flask OAuth2 for devices was created out of frustation of the last of good examples for building the device flow for OAuth 2.

### 1.1.2 Installation

This part of the documentation covers the installation of Flask-OAuthlib. The first step to using any software package is getting it properly installed.

#### Pip

Installing Flask-OAuth2-devices is simple with pip:

```
$ pip install flask-oauth2-devices
```

#### Get the Code

Flask-OAuth2-devices is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
git clone git://github.com/greedo/flask-oauth2-devices.git
```

### 1.1.3 OAuth2 Server

An OAuth2 server concerns how to grant the authorization and how to protect the resource. Register an **OAuth** provider:

```
from flask_oauth2_devices.provider import OAuth2DevicesProvider

app = Flask(__name__)
oauth = OAuth2DevicesProvider(app)
```

Like any other Flask extensions, we can pass the application later:

```
oauth = OAuth2DevicesProvider()

def create_app():
    app = Flask(__name__)
    oauth.init_app(app)
    return app
```

To implement the authorization flow, we need to understand the data model.

### Client (Application)

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 properties:

- 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

---

**Note:** The value of the scope parameter is expressed as a list of space- delimited, case-sensitive strings.

via: http://tools.ietf.org/html/rfc6749#section-3.3

---

An example of the data model in SQLAlchemy (SQLAlchemy is not required):

```
class Client(db.Model):
    client_id = db.Column(db.String(40), primary_key=True)
    client_secret = db.Column(db.String(55), nullable=False)

    user_id = db.Column(db.ForeignKey('user.id'))
    user = db.relationship('User')

    _redirect_uris = db.Column(db.Text)
    _default_scopes = db.Column(db.Text)

    @property
    def client_type(self):
```

```
        return 'public'

    @property
    def redirect_uris(self):
        if self._redirect_uris:
            return self._redirect_uris.split()
        return []

    @property
    def default_redirect_uri(self):
        return self.redirect_uris[0]

    @property
    def default_scopes(self):
        if self._default_scopes:
            return self._default_scopes.split()
        return []
```

### Configuration

The oauth provider has some built-in defaults, you can change them with Flask config:

| | |
|---|---|
| *OAUTH2_PROVIDER_ERROR_URI* | The error page when there is an error, default value is `'/oauth/errors'`. |
| *OAUTH2_PROVIDER_ERROR_ENDPOINT* | You can also configure the error page uri with an endpoint name. |
| *OAUTH2_PROVIDER_CODE_EXPIRES_IN* | Default OAuth code expires time, default is `3600`. |

### Implementation

The implementation of authorization flow needs two handlers, one is the code handler generate the initial user_code and device_code, the other is the authorization handler for the device to request an access token once the user has authorized the device.

Before the implementing of authorize and token handler, we need to set up some getters and setters to communicate with the database.

#### Client getter

A client getter is required. It tells which client is sending the requests, creating the getter with decorator:

```
@oauth.clientgetter
def load_client(client_id):
    return Client.query.filter_by(client_id=client_id).first()
```

#### Auth code getter and setter

Auth code getter and setter are required. They are used in the authorization flow, implemented with decorators:

```
@oauth.authcodegetter
def load_auth_code(code):
    return Code.query.filter_by(code=code).first()
```

In our example our auth code setter also creates creates new auth codes:

```
@oauth.authcodesetter
def save_auth_code(code, client_id, user_id, *args, **kwargs):

    expires_in = (AUTH_EXPIRATION_TIME if code is None else code.pop('expires_in'))
    expires = datetime.utcnow() + timedelta(seconds=expires_in)
    created = datetime.utcnow()

    cod = Code(
        client_id=client_id,
        user_id=user_id,
        code = (None if code is None else code['code']),
        _scopes = ('public private' if code is None else code['scope']),
        expires=expires,
        created=created,
        is_active=0
    )

    if cod.code is None:
        cod.code = cod.generate_new_code(cod.client_id)[:8]

    db.session.add(cod)
    db.session.commit()
    return cod
```

In the sample code, there is a `get_current_user` method, that will return the current user object, you should implement it yourself.

### Token creation

You are free to generate access tokens in whatever way you want. We have provided an example for creating access tokens and refresh tokens on the token object:

```
 def create_access_token(self, client_id, user_id, scope, token_type):

    expires_in = AUTH_EXPIRATION_TIME
    expires = datetime.utcnow() + timedelta(seconds=expires_in)
    created = datetime.utcnow()

    tok = Token(
        client_id=client_id,
        user_id=user_id,
        access_token=None,
        refresh_token=None,
        token_type=token_type,
        _scopes = ("public private" if scope is None else ' '.join(scope)),
        expires=expires,
        created=created,
    )

    if tok.access_token is None:
        tok.access_token = tok._generate_token()

    db.session.add(tok)
    db.session.commit()
    return tok

def refresh(self, token):
```

```
    tok = Token(
        client_id=self.client_id,
        user_id=self.user_id,
        access_token=self.access_token,
        refresh_token=None,
        token_type=token_type,
        _scopes = ("public private" if scope is None else ' '.join(scope)),
        expires=expires,
        created=created,
    )

    if tok.refresh_token is None:
        tok.refresh_token = tok._generate_refresh_token()

    db.session.add(tok)
    db.session.commit()
    return tok
```

The crytographic functions you use to generate the actual tokens are totally up to you, however we have some example in the example code.

#### Code handler

Code handler is a decorator for generating Auth Codes. You don't need to do much:

```
@app.route('/oauth/device', methods=['POST'])
@oauth.code_handler("https://api.example.com/oauth/device/authorize", "https://example.com/activate",
def code():
    return None
```

It expects the following parameters

- Authroize URL
- Activate URL
- Expires Internal
- Recommended Polling Internal

#### Authorize handler

Authorize handler is a decorator for the device to request an access token once the user has authorized the device. You don't need to do much:

```
@app.route('/oauth/device/authorize', methods=['POST'])
@oauth.authorize_handler()
def authorize():
    return None
```

#### Protect Resource

Protect the resource of a user with `require_oauth` decorator now:

```
@app.route('/api/me')
@oauth.require_oauth('email')
def me():
    user = request.oauth.user
    return jsonify(email=user.email, username=user.username)
```

### Example for OAuth 2 for devices

Here is an example of OAuth 2 server: https://github.com/greedo/flask-oauth2-devices/example

CHAPTER 2

# Additional Notes

Contribution guide and legal information are here.

## 2.1 authors

Flask-OAuthlib is written and maintained by Joe Cabrera <jcabrera@eminorlabs.com>.

### 2.1.1 Contributors

People who send patches and suggestions:

- Joe Cabrera http://github.com/greedo

## 2.2 License