oidcop

Release 0.1.0

Giuseppe De Marco, Roland Hedberg

May 13, 2022

INTRODUCTION

This project is a Python implementation of an **OIDC Provider** on top of jwtconnect.io that shows you how to 'build' an OP using the classes and functions provided by oidc-op.

If you are just going to build a standard OP you only have to write the configuration file. If you want to add or replace functionality this documentation should be able to tell you how.

Idpy OIDC-op implements the following standards:

- OpenID Connect Core 1.0 incorporating errata set 1
- Web Finger
- OpenID Connect Discovery 1.0 incorporating errata set 1
- OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1
- OpenID Connect Session Management 1.0
- OpenID Connect Back-Channel Logout 1.0
- OpenID Connect Front-Channel Logout 1.0
- OAuth2 Token introspection

It also comes with the following *add_on* modules.

- Custom scopes, that extends [OIDC standard ScopeClaims]
- Proof Key for Code Exchange by OAuth Public Clients (PKCE)
- OAuth2 PAR
- OAuth2 RAR
- OAuth2 DPoP

The entire project code is open sourced and therefore licensed under the Apache 2.0.

CHAPTER

ONE

THE OPENID CONNECT PROVIDER

1.1 Introduction

This documentation are here to show you how to 'build' an OP using the classes and functions provided by oidcop.

OAuth2 and thereby OpenID Connect (OIDC) are built on a request-response paradigm. The RP issues a request and the OP returns a response.

The OIDC core standard defines a set of such request-responses. This is a basic list of request-responses and the normal sequence in which they occur:

1. Provider discovery (WebFinger)
2. Provider Info Discovery
3. Client registration
4. Authorization/Authentication
5. Access token
6. User info

If you are just going to build a standard OP you only have to write the configuration file and of course add authentication and user consent services. If you want to add or replace functionality this document should be able to tell you how.

Setting up an OP means making a number if decisions. Like, should the OP support WebFinger , dynamic discovery and/or dynamic client registration .

All these are services you can access at endpoints. The total set of endpoints that this package supports are

- webfinger
- provider_info
- registration
- authorization
- token
- refresh_token
- userinfo
- end_session

1.2 Endpoint layout

When an endpoint receives a request it has to do a number of things:

- Verify that the client can issue the request (client authentication/authorization)
- Verify that the request is correct and that it contains the necessary information.
- Process the request, which includes applying server policies and gathering information.
- Construct the response

I should note at this point that this package is expected to work within the confines of a web server framework such that the actual receiving and sending of the HTTP messages are dealt with by the framework.

Based on the actions an endpoint has to perform a method call structure has been constructed. It looks like this:

1. parse_request
    - client_authentication (*)
    - post_parse_request (*)
2. process_request
3. do_response
    - **response_info**
        - **construct**
            * pre_construct (*)
            * _parse_args
            * post_construct (*)
    - update_http_args

Steps marked with '*' are places where extensions can be applied.

parse_request expects as input the request itself in a number of formats and also, if available, information about client authentication. The later is normally the authorization element of the HTTP header.

do_response returns a dictionary that can look like this:

```
{
  'response':
    _response as a string or as a Message instance_
  'http_headers': [
    ('Content-type', 'application/json'),
    ('Pragma', 'no-cache'),
    ('Cache-Control', 'no-store')
  ],
  'cookie': _list of cookies_,
  'response_placement': 'body'
}
```

cookie MAY be present

http_headers MAY be present

http_response Already clear and formatted HTTP response

response MUST be present

response_placement If absent defaults to the endpoints response_placement parameter value or if that is also missing 'url'

redirect_location Where to send a redirect

CHAPTER

TWO

SETUP

Create an environment

```
virtualenv -ppython3 env
source env/bin/activate
```

Install

```
pip install oidcop
```

Get the usage examples

```
git clone https://github.com/identitypython/oidc-op.git
cd oidc-op/example/flask_op/
bash run.sh
```

To configure a standard OIDC Provider you have to edit the oidcop configuration file. See `example/flask_op/config.json` to get in.

```
~/DEV/IdentityPython/OIDC/oidc-op/example/flask_op$ bash run.sh
2021-05-02 14:57:44,727 root DEBUG Configured logging using dictionary
2021-05-02 14:57:44,728 oidcop.configure DEBUG Set server password to {'kty': 'oct', 'use
↪': 'sig', 'k': 'n4G9OjOixYMOotXvP15grwq0peN2zq9I'}
 * Serving Flask app "oidc_op" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
2021-05-02 14:57:44,764 werkzeug INFO  * Running on https://127.0.0.1:5000/ (Press␣
↪CTRL+C to quit)
2021-05-02 14:57:44,765 werkzeug INFO  * Restarting with stat
2021-05-02 14:57:45,011 root DEBUG Configured logging using dictionary
2021-05-02 14:57:45,011 oidcop.configure DEBUG Set server password to {'kty': 'oct', 'use
↪': 'sig', 'k': 'bceYal7bK9zvlBAA7-23lsi5crcv_8Cd'}
2021-05-02 14:57:45,037 werkzeug WARNING  * Debugger is active!
2021-05-02 14:57:45,092 werkzeug INFO  * Debugger PIN: 560-973-597
```

Then open your browser to `https://127.0.0.1:5000/.well-known/openid-configuration` to get the OpenID Provider Configuration resource.

CHAPTER

THREE

JWK SET (JWKS) FILES

see: cryptojwt documentation

You can use `cryptojwt.key_jar.init_key_jar` to create JWKS file. An easy way can be to configure the auto creation of JWKS files directly in your conf.yaml file. Using `read_only: False` in `OIDC_KEYS` it will create the path within the JWKS files. Change it to `True` if you don't want to overwrite them on each execution.

In genral configuration:

```
OIDC_KEY_DEFS = [
    {
      "type": "RSA",
      "use": [
        "sig"
      ]
    },
    {
      "type": "EC",
      "crv": "P-256",
      "use": [
        "sig"
      ]
    }
]

OIDCOP_CONF = {
  "port": PORT,
  "domain": DOMAIN,
  "server_name": SERVER_NAME,
  "base_url": f"https://{SERVER_NAME}",
  "keys": {
    "private_path": "data/oidc_op/private/jwks.json",
    "key_defs": OIDC_KEY_DEFS,
    "public_path": "data/static/jwks.json",
    "read_only": False,
    "uri_path": "static/jwks.json"
  },
```

In the JWTConnect-Python-CryptoJWT distribution there is also a script you can use to construct a JWK. You can for instance do:

```
$ jwkgen --kty=RSA
{
```

(continues on next page)

(continued from previous page)

```
    "d": "b9ucfay9vxDvz_nRZMVSUR9eRvHNMo0tc8Bl7tWkwxTis7LBXxmbMH1yzLs8omUil_u2a-Z_
↪6VlKENxacuejYYcOhs6bfaU3iOqJbGi2p4t2i1oxjuF-cX6BZ5aHB5Wfb1uTXXobHokjcjVVDmBr_
↪fNYBEPtZsVYqyN9sR9KE_ZLHEPks3IER09aX9G3wiB_PgcxQDRAl72qucsBz9_W9KS-TVWs-
↪qCEqtXLmx9AAN6P8SjUcHAzEb0ZCJAYCkVu34wgNjxVaGyYN1qMA-1iOOVz--wtMyBwc5atSDBDgUApxFyj_
↪DHSeBl81IHedcPjS9azxqFhumP7oJJyfecfSQ",
    "e": "AQAB",
    "kid": "cHZQbWRrMzRZak53U1pfSUNjY0dKd2xXaXRKenktdUduUjVBVTl3VE5ndw",
    "kty": "RSA",
    "n": "73XCXV2iiubSCEaFe26OpVnsBFlXwXh_yDCDyBqFgAFi5WdZTpRMJZoK0nn_
↪vv2MvrXqFnw6IfXkwdsRGlMsNldVy36003gKa584CNksxfenwJZcF-huASUrSJEFr-3c0fMT_
↪pLyAc7yf3rNCdRegzbBXSvIGKQpaeIjIFYftAPd9tjGA_
↪SuYWVQDsSh3MeGbB4wt0lArAyFZ4f5o7SSxSDRCUF3ng3CB_QKUAaDHHgXrcNG_
↪gPpgqQZjsDJ0VwMXjFKxQmskbH-dfsQ05znQsYn3pjcd_TEZ-Yu765_L5uxUrkEy_
↪KnQXe1iqaQHcnfBWKXt18NAuBfgmKsv8gnxQ",
    "p": "_RPgbiQcFu8Ekp-tC-Kschpag9iaLc9aDqrxE6GWuThEdExGngP_
↪p1I7Qd7gXHHTMXLp1c4gH2cKx4AkfQyKny2RJGtV2onQButUU5r0gwnlqqycIA2Dc9JiH85PX2Z889TKJUlVETfYbezHbKhdsazjjs
↪",
    "q":
↪"8jmgnadtwjMt96iOaoL51irPRXONO82tLM2AAZAK5Obsj23bZ9LFiw2Joh5oCSFdoUcRhbbIhCIv2aT4T_
↪XKnDGnddrkxpF5Xgu0-hPNYnJx5m4kuzerot4j79Tx6qO-bshaaGz50MHs1vHSeFaDVN4fvh_
↪hDWpV1BCNI0PKK-c"
}
SHA-256: pvPmdk34YjNwSZ_ICccGJwlWitJzy-uGnR5AU9wTNgw
```

Example: create a JWK for cookie signing

```
jwkgen --kty=SYM --kid cookie > private/cookie_sign_jwk.json
```

CHAPTER

FOUR

CONFIGURATION DIRECTIVES

4.1 issuer

The issuer ID of the OP, a unique value in URI format.

4.2 session params

Configuration parameters used by session manager:

```
"session_params": {
  "password": "__password_used_to_encrypt_access_token_sid_value",
  "salt": "salt involved in session sub hash ",
  "sub_func": {
    "public": {
      "class": "oidcop.session.manager.PublicID",
      "kwargs": {
        "salt": "sdfsdfdsf"
      }
    },
    "pairwise": {
      "class": "oidcop.session.manager.PairWiseID",
      "kwargs": {
        "salt": "sdfsdfsdf"
      }
    }
 }
},
```

4.2.1 password

Optional. Encryption key used to encrypt the SessionID (sid) in access_token. If unset it will be random.

4.2.2 salt

Optional. Salt, value or filename, used in sub_funcs (pairwise, public) for creating the opaque hash of *sub* claim.

4.2.3 sub_funcs

Optional. Functions involved in subject value creation.

4.2.4 scopes_to_claims

A dict defining the scopes that are allowed to be used per client and the claims they map to (defaults to the scopes mapping described in the spec). If we want to define a scope that doesn't map to claims (e.g. offline_access) then we simply map it to an empty list. E.g.:

```
{
    "scope_a": ["claim1", "claim2"],
    "scope_b": []
}
```

Note: For OIDC the *openid* scope must be present in this mapping.

4.2.5 allowed_scopes

A list with the scopes that are allowed to be used (defaults to the keys in scopes_to_claims).

4.2.6 scopes_supported

A list with the scopes that will be advertised in the well-known endpoint (defaults to allowed_scopes).

4.3 add_on

An example:

```
"add_on": {
    "pkce": {
      "function": "oidcop.oidc.add_on.pkce.add_pkce_support",
      "kwargs": {
        "essential": false,
        "code_challenge_method": "S256 S384 S512"
      }
    },
  }
```

The provided add-ons can be seen in the following sections.

4.3.1 pkce

The pkce add on is activated using the `oidcop.oidc.add_on.pkce.add_pkce_support` function. The possible configuration options can be found below.

essential

Whether pkce is mandatory, authentication requests without a `code_challenge` will fail if this is True. This option can be overridden per client by defining `pkce_essential` in the client metadata.

code_challenge_method

The allowed code_challenge methods. The supported code challenge methods are: `plain, S256, S384, S512`

4.4 authentication

An example:

```
"authentication": {
    "user": {
      "acr": "urn:oasis:names:tc:SAML:2.0:ac:classes:InternetProtocolPassword",
      "class": "oidcop.user_authn.user.UserPassJinja2",
      "kwargs": {
        "verify_endpoint": "verify/user",
        "template": "user_pass.jinja2",
        "db": {
          "class": "oidcop.util.JSONDictDB",
          "kwargs": {
            "filename": "passwd.json"
          }
        },
        "page_header": "Testing log in",
        "submit_btn": "Get me in!",
        "user_label": "Nickname",
        "passwd_label": "Secret sauce"
      }
    }
  },
```

4.5 capabilities

This covers most of the basic functionality of the OP. The key words are the same as defined in OIDC Discovery. A couple of things are defined else where. Like the endpoints, issuer id, jwks_uri and the authentication methods at the token endpoint.

An example:

```
response_types_supported:
    - code
    - token
    - id_token
    - "code token"
    - "code id_token"
    - "id_token token"
    - "code id_token token"
    - none
  response_modes_supported:
    - query
    - fragment
    - form_post
  subject_types_supported:
    - public
    - pairwise
  grant_types_supported:
    - authorization_code
    - implicit
    - urn:ietf:params:oauth:grant-type:jwt-bearer
    - refresh_token
  claim_types_supported:
    - normal
    - aggregated
    - distributed
  claims_parameter_supported: True
  request_parameter_supported: True
  request_uri_parameter_supported: True
  frontchannel_logout_supported: True
  frontchannel_logout_session_supported: True
  backchannel_logout_supported: True
  backchannel_logout_session_supported: True
  check_session_iframe: https://127.0.0.1:5000/check_session_iframe
  scopes_supported: ["openid", "profile", "random"]
  claims_supported: ["sub", "given_name", "birthdate"]
```

4.6 client_db

If you're running an OP with static client registration you want to keep the registered clients in a database separate from the session database since it will change independent of the OP process. In this case you need *client_db*. If you are on the other hand only allowing dynamic client registration then keeping registered clients only in the session database makes total sense.

The class you reference in the specification MUST be a subclass of oidcmsg.storage.DictType and have some of the methods a dictionary has.

Note also that this class MUST support the dump and load methods as defined in `oidcmsg.impexp.ImpExp`.

An example:

```
client_db: {
    "class": 'oidcmsg.abfile.AbstractFileSystem',
```

(continues on next page)

(continued from previous page)

```
    "kwargs": {
        'fdir': full_path("afs"),
        'value_conv': 'oidcmsg.util.JSON'
    }
}
```

4.7 cookie_handler

An example:

```
  "cookie_handler": {
    "class": "oidcop.cookie_handler.CookieHandler",
    "kwargs": {
      "keys": {
        "private_path": f"{OIDC_JWKS_PRIVATE_PATH}/cookie_jwks.json",
        "key_defs": [
          {"type": "OCT", "use": ["enc"], "kid": "enc"},
          {"type": "OCT", "use": ["sig"], "kid": "sig"}
        ],
        "read_only": False
      },
      "flags": {
          "samesite": "None",
          "httponly": True,
          "secure": True,
      },
      "name": {
        "session": "oidc_op",
        "register": "oidc_op_rp",
        "session_management": "sman"
      }
    }
},
```

4.8 endpoint

An example:

```
"endpoint": {
  "webfinger": {
    "path": ".well-known/webfinger",
    "class": "oidcop.oidc.discovery.Discovery",
    "kwargs": {
      "client_authn_method": null
    }
  },
  "provider_info": {
    "path": ".well-known/openid-configuration",
```

(continues on next page)

(continued from previous page)

```
    "class": "oidcop.oidc.provider_config.ProviderConfiguration",
    "kwargs": {
      "client_authn_method": null
    }
  },
  "registration": {
    "path": "registration",
    "class": "oidcop.oidc.registration.Registration",
    "kwargs": {
      "client_authn_method": None,
      "client_secret_expiration_time": 432000,
      "client_id_generator": {
         "class": 'oidcop.oidc.registration.random_client_id',
         "kwargs": {
              "seed": "that-optional-random-value"
         }
     }
    }
  },
  "registration_api": {
    "path": "registration_api",
    "class": "oidcop.oidc.read_registration.RegistrationRead",
    "kwargs": {
      "client_authn_method": [
        "bearer_header"
      ]
    }
  },
  "introspection": {
    "path": "introspection",
    "class": "oidcop.oauth2.introspection.Introspection",
    "kwargs": {
      "client_authn_method": [
        "client_secret_post",
        "client_secret_basic",
        "client_secret_jwt",
        "private_key_jwt"
      ]
      "release": [
        "username"
      ]
    }
  },
  "authorization": {
    "path": "authorization",
    "class": "oidcop.oidc.authorization.Authorization",
    "kwargs": {
      "client_authn_method": null,
      "claims_parameter_supported": true,
      "request_parameter_supported": true,
      "request_uri_parameter_supported": true,
      "response_types_supported": [
```

(continues on next page)

(continued from previous page)

```
        "code",
        "token",
        "id_token",
        "code token",
        "code id_token",
        "id_token token",
        "code id_token token",
        "none"
      ],
      "response_modes_supported": [
        "query",
        "fragment",
        "form_post"
      ]
    }
  },
  "token": {
    "path": "token",
    "class": "oidcop.oidc.token.Token",
    "kwargs": {
      "client_authn_method": [
        "client_secret_post",
        "client_secret_basic",
        "client_secret_jwt",
        "private_key_jwt",
      ],
      "revoke_refresh_on_issue": True
    }
  },
  "userinfo": {
    "path": "userinfo",
    "class": "oidcop.oidc.userinfo.UserInfo",
    "kwargs": {
      "claim_types_supported": [
        "normal",
        "aggregated",
        "distributed"
      ]
    }
  },
  "end_session": {
    "path": "session",
    "class": "oidcop.oidc.session.Session",
    "kwargs": {
      "logout_verify_url": "verify_logout",
      "post_logout_uri_path": "post_logout",
      "signing_alg": "ES256",
      "frontchannel_logout_supported": true,
      "frontchannel_logout_session_supported": true,
      "backchannel_logout_supported": true,
      "backchannel_logout_session_supported": true,
      "check_session_iframe": "check_session_iframe"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

You can specify which algoritms are supported, for example in userinfo_endpoint:

```
"userinfo_signing_alg_values_supported": OIDC_SIGN_ALGS,
"userinfo_encryption_alg_values_supported": OIDC_ENC_ALGS,
```

Or in authorization endpoint:

```
"request_object_encryption_alg_values_supported": OIDC_ENC_ALGS,
```

4.9 httpc_params

Parameters submitted to the web client (python requests). In this case the TLS certificate will not be verified, to be intended exclusively for development purposes

Example

```
"httpc_params": {
    "verify": false
  },
```

4.10 keys

An example:

```
"keys": {
    "private_path": "private/jwks.json",
    "key_defs": [
      {
        "type": "RSA",
        "use": [
          "sig"
        ]
      },
      {
        "type": "EC",
        "crv": "P-256",
        "use": [
          "sig"
        ]
      }
    ],
    "public_path": "static/jwks.json",
    "read_only": false,
    "uri_path": "static/jwks.json"
  },
```

read_only means that on each restart the keys will created and overwritten with new ones. This can be useful during the first time the project have been executed, then to keep them as they are *read_only* would be configured to *True*.

4.11 login_hint2acrs

OIDC Login hint support, it's optional. It matches the login_hint paramenter to one or more Authentication Contexts.

An example:

```
"login_hint2acrs": {
  "class": "oidcop.login_hint.LoginHint2Acrs",
  "kwargs": {
    "scheme_map": {
      "email": [
        "urn:oasis:names:tc:SAML:2.0:ac:classes:InternetProtocolPassword"
      ]
    }
  }
},
```

oidc-op supports the following authn contexts:

- UNSPECIFIED, urn:oasis:names:tc:SAML:2.0:ac:classes:unspecified
- INTERNETPROTOCOLPASSWORD, urn:oasis:names:tc:SAML:2.0:ac:classes:InternetProtocolPassword
- MOBILETWOFACTORCONTRACT, urn:oasis:names:tc:SAML:2.0:ac:classes:MobileTwoFactorContract
- PASSWORDPROTECTEDTRANSPORT, urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
- PASSWORD, urn:oasis:names:tc:SAML:2.0:ac:classes:Password
- TLSCLIENT, urn:oasis:names:tc:SAML:2.0:ac:classes:TLSClient
- TIMESYNCTOKEN, urn:oasis:names:tc:SAML:2.0:ac:classes:TimeSyncToken

4.12 authz

This configuration section refers to the authorization/authentication endpoint behaviour. Scopes bound to an access token are strictly related to grant management, as part of what that endpoint does. Regarding grant authorization we should have something like the following example.

If you omit this section from the configuration (thus using some sort of default profile) you'll have an Implicit grant authorization that leads granting nothing. Add the below to your configuration and you'll see things changing.

An example:

```
"authz": {
  "class": "oidcop.authz.AuthzHandling",
  "kwargs": {
      "grant_config": {
          "usage_rules": {
              "authorization_code": {
                  "supports_minting": ["access_token", "refresh_token", "id_token"],
```

(continues on next page)

(continued from previous page)

```
                    "max_usage": 1
                },
                "access_token": {},
                "refresh_token": {
                    "supports_minting": ["access_token", "refresh_token"]
                }
            },
            "expires_in": 43200
        }
    }
},
```

4.13 template_dir

The HTML Template directory used by Jinja2, used by endpoint context template loader, as:

```
Environment(loader=FileSystemLoader(template_dir), autoescape=True)
```

An example:

```
"template_dir": "templates"
```

For any further customization of template here an example of what used in django-oidc-op:

```
"authentication": {
  "user": {
    "acr": "urn:oasis:names:tc:SAML:2.0:ac:classes:InternetProtocolPassword",
    "class": "oidc_provider.users.UserPassDjango",
    "kwargs": {
      "verify_endpoint": "verify/oidc_user_login/",
      "template": "oidc_login.html",

      "page_header": "Testing log in",
      "submit_btn": "Get me in!",
      "user_label": "Nickname",
      "passwd_label": "Secret sauce"
    }
  }
},
```

4.14 token_handler_args

Token handler is an intermediate interface used by and endpoint to manage the tokens' default behaviour, like lifetime and minting policies. With it we can create a token that's linked to another, and keep relations between many tokens in session and grants management.

An example:

```
"token_handler_args": {
    "jwks_def": {
      "private_path": "private/token_jwks.json",
      "read_only": false,
      "key_defs": [
        {
          "type": "oct",
          "bytes": 24,
          "use": [
            "enc"
          ],
          "kid": "code"
        },
        {
          "type": "oct",
          "bytes": 24,
          "use": [
            "enc"
          ],
          "kid": "refresh"
        }
      ]
    },
    "code": {
      "kwargs": {
        "lifetime": 600
      }
    },
    "token": {
      "class": "oidcop.token.jwt_token.JWTToken",
      "kwargs": {
          "lifetime": 3600,
          "add_claims": [
            "email",
            "email_verified",
            "phone_number",
            "phone_number_verified"
          ],
          "add_claims_by_scope": true,
          "aud": ["https://example.org/appl"]
       }
    },
    "refresh": {
        "kwargs": {
            "lifetime": 86400
        }
    }
    "id_token": {
        "class": "oidcop.token.id_token.IDToken",
        "kwargs": {
            "base_claims": {
                "email": None,
                "email_verified": None,
```

(continues on next page)

(continued from previous page)

```
        },
    }
  }
```

jwks_defs can be replaced eventually by *jwks_file*:

```
"jwks_file": f"{OIDC_JWKS_PRIVATE_PATH}/token_jwks.json",
```

You can even select wich algorithms to support in id_token, eg:

```
"id_token": {
    "class": "oidcop.token.id_token.IDToken",
    "kwargs": {
        "id_token_signing_alg_values_supported": [
                "RS256",
                "RS512",
                "ES256",
                "ES512",
                "PS256",
                "PS512",
            ],
        "id_token_encryption_alg_values_supported": [
                "RSA-OAEP",
                "RSA-OAEP-256",
                "A192KW",
                "A256KW",
                "ECDH-ES",
                "ECDH-ES+A128KW",
                "ECDH-ES+A192KW",
                "ECDH-ES+A256KW",
            ],
        "id_token_encryption_enc_values_supported": [
                'A128CBC-HS256',
                'A192CBC-HS384',
                'A256CBC-HS512',
                'A128GCM',
                'A192GCM',
                'A256GCM'
            ],
    }
}
```

4.15 userinfo

An example:

```
"userinfo": {
    "class": "oidcop.user_info.UserInfo",
    "kwargs": {
      "db_file": "users.json"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

This is somethig that can be customized. For example in the django-oidc-op implementation is used something like the following:

```
"userinfo": {
    "class": "oidc_provider.users.UserInfo",
    "kwargs": {
        "claims_map": {
            "phone_number": "telephone",
            "family_name": "last_name",
            "given_name": "first_name",
            "email": "email",
            "verified_email": "email",
            "gender": "gender",
            "birthdate": "get_oidc_birthdate",
            "updated_at": "get_oidc_lastlogin"
        }
    }
}
```

CHAPTER

FIVE

CLIENTS

In this section there are some client configuration examples. That can be used to override the global configuration of the OP.

How to configure the release of the user claims per clients:

```
endpoint_context.cdb["client_1"] = {
    "client_secret": "hemligt",
    "redirect_uris": [("https://example.com/cb", None)],
    "response_types": ["code", "token", "code id_token", "id_token"],
    "add_claims": {
        "always": {
            "introspection": ["nickname", "eduperson_scoped_affiliation"],
            "userinfo": ["picture", "phone_number"],
        },
        # this overload the general endpoint configuration for this client
        # self.server.server_get("endpoint", "id_token").kwargs = {"add_claims_by_scope
↪": True}
        "by_scope": {
            "id_token": False,
        },
    },
```

The available configuration options are:

5.1 client_secret

The client secret. This parameter is required.

5.2 client_secret_expires_at

When the client_secret expires.

5.3 redirect_uris

The client's redirect uris.

5.4 auth_method

The auth_method that can be used per endpoint. E.g:

```
{
    "AccessTokenRequest": "client_secret_basic",
    ...
}
```

5.5 request_uris

A list of *request_uris*.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata.

5.6 response_types

The allowed *response_types* for this client.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata.

5.7 grant_types_supported

Configure the allowed grant types on the token endpoint.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata.

5.8 scopes_to_claims

A dict defining the scopes that are allowed to be used per client and the claims they map to (defaults to the scopes mapping described in the spec). If we want to define a scope that doesn't map to claims (e.g. offline_access) then we simply map it to an empty list. E.g.:

```
{
  "scope_a": ["claim1", "claim2"],
  "scope_b": []
}
```

5.9 allowed_scopes

A list with the scopes that are allowed to be used (defaults to the keys in the clients scopes_to_claims).

5.10 revoke_refresh_on_issue

Configure whether to revoke the refresh token that was used to issue a new refresh token.

5.11 add_claims

A dictionary with the following keys

5.11.1 always

A dictionary with the following keys: *userinfo*, *id_token*, *introspection*, *access_token*. The keys are used to describe the claims we want to add to the corresponding interface. The keys can be a list of claims to be added or a dict in the format described in https://openid.net/specs/openid-connect-core-1_0.html#IndividualClaimsRequests E.g.:

```
{
    "add_claims": {
        "always": {
          "userinfo": ["email", "phone"], # Always add "email" and "phone" in the␣
↪userinfo response if such claims exists
          "id_token": {"email": null}, # Always add "email" in the id_token if such a␣
↪claim exists
          "introspection": {"email": {"value": "a@a.com"}}, # Add "email" in the␣
↪introspection response only if its value is "a@a.com"
        }
    }
}
```

5.11.2 by_scope

A dictionary with the following keys: *userinfo*, *id_token*, *introspection*, *access_token*. The keys are boolean values that describe whether the scopes should be mapped to claims and added to the response. E.g.:

```
{
    "add_claims": {
        "by_scope": {
            id_token: True, # Map the requested scopes to claims and add them to the id␣
↪token
}
```

5.12 token_usage_rules

The usage rules for each token type. E.g.:

```
{
    "usage_rules": {
        "authorization_code": {
            "expires_in": 3600,
            "supports_minting": [
                "access_token",
                "id_token",
            ],
            "max_usage": 1,
        },
        "access_token": {
            "expires_in": self.params["access_token_lifetime"],
        },
    }
}
```

5.13 pkce_essential

Whether pkce is essential for this client.

5.14 post_logout_redirect_uri

The client's post logout redirect uris.

See https://openid.net/specs/openid-connect-rpinitiated-1_0.html#RPLogout.

5.15 backchannel_logout_uri

The client's *backchannel_logout_uri*.

See https://openid.net/specs/openid-connect-backchannel-1_0.html#BCRegistration

5.16 frontchannel_logout_uri

The client's *frontchannel_logout_uri*.

See https://openid.net/specs/openid-connect-frontchannel-1_0.html#RPLogout

5.17 request_object_signing_alg

A list with the allowed algorithms for signing the request object.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.18 request_object_encryption_alg

A list with the allowed alg algorithms for encrypting the request object.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.19 request_object_encryption_enc

A list with the allowed enc algorithms for signing the request object.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.20 userinfo_signed_response_alg

JWS alg algorithm [JWA] REQUIRED for signing UserInfo Responses.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.21 userinfo_encrypted_response_enc

The alg algorithm [JWA] REQUIRED for encrypting UserInfo Responses.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.22 userinfo_encrypted_response_alg

JWE enc algorithm [JWA] REQUIRED for encrypting UserInfo Responses.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.23 id_token_signed_response_alg

JWS alg algorithm [JWA] REQUIRED for signing ID Token issued to this Client.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.24 id_token_encrypted_response_enc

The alg algorithm [JWA] REQUIRED for encrypting ID Token issued to this Client.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.25 id_token_encrypted_response_alg

JWE enc algorithm [JWA] REQUIRED for encrypting ID Token issued to this Client.

See https://openid.net/specs/openid-connect-registration-1_0-29.html#ClientMetadata

5.26 dpop_jkt

CHAPTER

SIX

USAGE

Some examples, how to run flask_op and django_op but also some typical configuration in relation to common use cases.

CHAPTER

SEVEN

CONFIGURE FLASK-RP

JWTConnect-Python-OidcRP is Relaing Party for tests, see related page. You can run a working instance of `JWTConnect-Python-OidcRP.flask_rp` with:

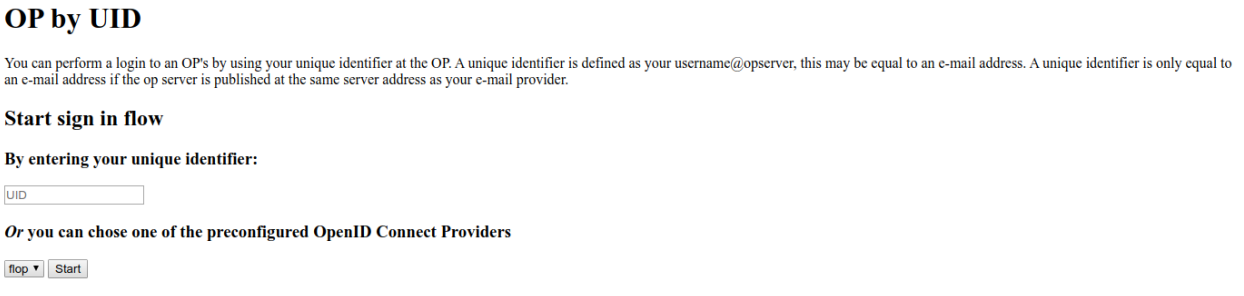
```
pip install git+https://github.com/openid/JWTConnect-Python-OidcRP.git

# get entire project to have examples files
git clone https://github.com/openid/JWTConnect-Python-OidcRP.git
cd JWTConnect-Python-OidcRP/example/flask_rp

# run it as it come
bash run.sh
```

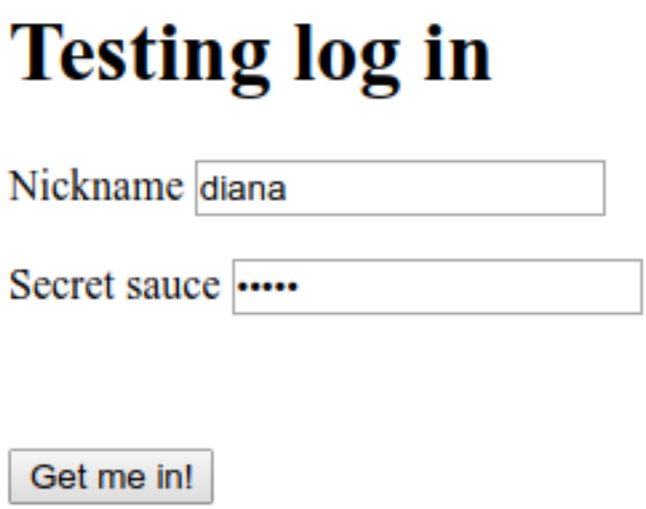
Now you can connect to `https://127.0.0.1:8090/` to see the RP landing page and select your authentication endpoint.

7.1 Authentication examples



RP

Get to the RP landing page to choose your authentication endpoint. The first option aims to use *Provider Discovery*.



OP Auth

The AS/OP supports dynamic client registration, it accepts the authentication request and prompt to us the login form. Read passwd.json file to get credentials.

OP result

You have successfully logged in!

Accesstoken
eyJhbGciOiJFUzI1NiIsImtpZCI6IlpFdElNMjlFTjBWc2J6QlRaRE5NTUdaRGRuaDRWa1p0T0hkTmVXdENlUzFQVDFl
Cz9plr3TuA

Endpoints

Authorization endpoint
https://127.0.0.1:5000/authorization
Token endpoint
https://127.0.0.1:5000/token
Userinfo endpoint
https://127.0.0.1:5000/userinfo
End session endpoint
https://127.0.0.1:5000/session
Registration endpoint
https://127.0.0.1:5000/registration

User information

nickname
Dina
phone_number
+46 90 7865000
address
{'street_address': 'Umeå Universitet', 'country': 'Sweden', 'locality': 'Umeå', 'postal_code': 'SE-90187'}
email_verified
False
sub
1b2fc9341a16ae4e30082965d537ae47c21a0f27fd43eab78330ed81751ae6db
email
diana@example.org
given_name
Diana
name
Diana Krall
family_name
Krall

Logout!

Access

The identity representation with the information fetched from the user info endpoint.

Logout

We can even test the single logout

CHAPTER

EIGHT

REFRESH TOKEN

To obtain a refresh token, you have to use `response_type=code`, add `offline_access` to `scope` and also use `prompt=consent`, otherwise there will be an error (based on OpenID Connect specification).

To refresh a token:

```
import requests

CLIENT_ID = "DBP60x3KUQfCYWZlqFaS_Q"
CLIENT_SECRET="8526270403788522b2444e87ea90c53bcafb984119cec92eeccc12f1"
REFRESH_TOKEN = "Z0FBQUFBQ ... lN2JNODYtZThjMnFsZUNDcg=="

data = {
    "grant_type" : "refresh_token",
    "client_id" : f"{CLIENT_ID}",
    "client_secret" : f"{CLIENT_SECRET}",
    "refresh_token" : f"{REFRESH_TOKEN}"
}
headers = {'Content-Type': "application/x-www-form-urlencoded" }
response = requests.post(
    'https://127.0.0.1:8000/oidcop/token', verify=False, data=data, headers=headers
)
```

oidc-op will return a json response like this:

```
{
 'access_token': 'eyJhbGc ... CIOH_09tT_YVa_gyTqg',
 'token_type': 'Bearer',
 'scope': 'openid profile email address phone offline_access',
 'refresh_token': 'Z0FBQ ... 1TE16cm1Tdg=='
}
```

CHAPTER

NINE

INTROSPECTION ENDPOINT

Here an example about how to consume oidc-op introspection endpoint. This example uses a client with an HTTP Basic Authentication::

```
import base64
import requests

TOKEN =
↪"eyJhbGciOiJFUzI1NiIsImtpZCI6IlQwZGZTM1ZVYUcxS1ZubG9VVTQwUXpJMlMyMHpjSHBRYlMxdGIzZ3hZVWhCYzNGaFZWTlpT
↪eyJzY29wZSI6IFsib3BlbmlkIiwgInByb2ZpbGUiLCAiZW1haWwiLCAiYWRkcmVzcyIsICJwaG9uZSJdLCAiYXVkIjogWyJvTHlSa
↪pVqxUNznsoZu9ND18IEMJIHDOT6_HxzoFiTLsniNdbAdXTuOoiaKeRTqtDyjT9WuUPszdHkVjt5xxeFX8gQMuA"

data = {
 'token': TOKEN,
 'token_type_hint': 'access_token'
}

_basic_secret = base64.b64encode(
    f'{"oLyRj7sJJ3XvAYjeDCe8rQ"}:{
↪"53fb49f2a6501ec775355c89750dc416744a3253138d5a04e409b313"}'.encode()
)
headers = {
    'Authorization': f"Basic {_basic_secret.decode()}"
}

requests.post('https://127.0.0.1:8000/introspection', verify=False, data=data,␣
↪headers=headers)
```

oidc-op will return a json response like this::

```
{
  "active": true,
  "scope": "openid profile email address phone",
  "client_id": "oLyRj7sJJ3XvAYjeDCe8rQ",
  "token_type": "access_token",
  "exp": 0,
  "iat": 1621777305,
  "sub": "a7b0dea2958aec275a789d7d7dc8e7d09c6316dd4fc6ae92742ed3297e14dded",
  "iss": "https://127.0.0.1:8000",
  "aud": [
    "oLyRj7sJJ3XvAYjeDCe8rQ"
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

CHAPTER

TEN

SESSION MANAGEMENT

10.1 About session management

The OIDC Session Management draft defines session to be:

> Continuous period of time during which an End-User accesses a Relying Party relying on the Authentication of the End-User performed by the OpenID Provider.

Note that we are dealing with a Single Sign On (SSO) context here. If for some reason the OP does not want to support SSO then the session management has to be done a bit differently. In that case each session (user_id,client_id) would have its own authentication event. Not one shared between the sessions.

10.1.1 Design criteria

So a session is defined by a user and a Relying Party. If one adds to that that a user can have several sessions active at the same time each one against a unique Relying Party we have the bases for session management.

Furthermore the user may well decide on different rules for different relying parties for releasing user attributes, where and how issued access tokens could be used and whether refresh tokens should be issued or not.

We also need to keep track on which tokens where used to mint new tokens such that we can easily revoked a suite of tokens all with a common ancestor.

10.1.2 Database layout

The database is organized in 3 levels. The top one being the users. Below that the Relying Parties and at the bottom what is called grants.

Grants organize authorization codes, access tokens and refresh tokens (and possibly other types of tokens) in a comprehensive way. More about that below.

There may be many Relying Parties below a user and many grants below a Relying Party.

10.2 The information structure

As stated above there are 3 layers: user session information, client session information and grants. But first the keys to the information.

10.2.1 Session key

A key to the session information is based on a list. The first item being the user identifier, the second the client identifier and the third the grant identifier. If you only want the user session information then the key is a list with one item the user id. If you want the client session information the key is a list with 2 items (user_id, client_id). And lastly if you want a grant then the key is a list with 3 elements (user_id, client_id, grant_id).

Example:: "diana;;KtEST70jZx1x;;85544c9cace411ebab53559c5425fcc0"

A *session identifier* is constructed using the **session_key** function. It takes as input the 3 elements list.:

```
session_id = session_key(user_id, client_id, grant_id)
```

Using the function **unpack_session_key** you can get the elements from a session_id.:

```
user_id, client_id, grant_id = unpack_session_id(session_id)
```

10.2.2 User session information

Houses the authentication event information which is the same for all session connected to a user. Here we also have a list of all the clients that this user has a session with. Expressed as a dictionary this can look like this:

```
{
    'authentication_event': {
        'uid': 'diana',
        'authn_info': "urn:oasis:names:tc:SAML:2.0:ac:classes:InternetProtocolPassword",
        'authn_time': 1605515787,
        'valid_until': 1605519387
    },
    'subordinate': ['client_1']
}
```

10.2.3 Client session information

The client specific information of the session information. Presently only the authorization request and the subject identifier (sub). The subordinates to this set of information are the grants:

```
{
    'authorization_request':{
        'client_id': 'client_1',
        'redirect_uri': 'https://example.com/cb',
        'scope': ['openid', 'research_and_scholarship'],
        'state': 'STATE',
        'response_type': ['code']
    },
    'sub': '117afe8d7bb0ace8e7fb2706034ab2d3fbf17f0fd4c949aa9c23aedd051cc9e3',
    'subordinate': ['e996c61227e711eba173acde48001122'],
    'revoked': False
}
```

10.2.4 Grant information

Grants are created by an authorization subsystem in an OP. If the grant is created in connection with an user authentication the authorization system might normally ask the user for usage consent and then base the construction of the grant on that consent.

If an authorization server can act as a Security Token Service (STS) as defined by Token Exchange [RFC-8693] then no user is involved. In the context of session management the STS is equivalent to a user.

Grant information contains information about user consent and issued tokens.:

```
{
    "type": "grant",
    "scope": ["openid", "research_and_scholarship"],
    "authorization_details": null,
```

(continues on next page)

(continued from previous page)


```
    "claims": {
        "userinfo": {
            "sub": null,
            "name": null,
            "given_name": null,
            "family_name": null,
            "email": null,
            "email_verified": null,
            "eduperson_scoped_affiliation": null
        }
    },
    "resources": ["client_1"],
    "issued_at": 1605452123,
    "not_before": 0,
    "expires_at": 0,
    "revoked": false,
    "issued_token": [
        {
            "type": "authorization_code",
            "issued_at": 1605452123,
            "not_before": 0,
            "expires_at": 1605452423,
            "revoked": false,
            "value":
↪"Z0FBQUFBQmZzVUZieDFWZy1fbjE2ckxvZkFTVC1ZTHJIVlk0Z09tOVk1M0RsOVNDbkdfLTIxTUhILWs4T29kM1lmV015UEN1UGxr
↪",
            "usage_rules": {
                "expires_in": 300,
                "supports_minting": [
                    "access_token",
                    "refresh_token",
                    "id_token"
                ],
                "max_usage": 1
                },
            "used": 0,
            "based_on": null,
            "id": "96d19bea275211eba43bacde48001122"
        },
        {
            "type": "access_token",
            "issued_at": 1605452123,
            "not_before": 0,
            "expires_at": 1605452723,
            "revoked": false,
            "value":
↪"Z0FBQUFBQmZzVUZiaWVRbi1IS2k0VW4wVDY1ZmJHeEVCR1hVODBaQXR6MWkzelNBRFpOS2tRM3p4WWY5Y1J6dk5IWWpnelRETGVp
↪",
            "usage_rules": {
                "expires_in": 600,
            },
            "used": 0,
```


(continues on next page)

(continued from previous page)

```
            "based_on":
↪"Z0FBQUFBQmZzVUZieDFWZy1fbjE2ckxvZkFTVC1ZTHJIVlk0Z09tOVk1M0RsOVNDbkdfLTIxTUhILWs4T29kM1lmV015UEN1UGxr
↪",
            "id": "96d1c840275211eba43bacde48001122"
        }
    ],
    "id": "96d16d3c275211eba43bacde48001122"
}
```

The parameters are described below

scope

This is the scope that was chosen for this grant. Either by the user or by some rules that the Authorization Server runs by.

authorization_details

Presently a place hold. But this is expected to be information on how the authorization was performed. What input was used and so on.

claims

The set of claims that should be returned in different circumstances. The syntax that is defined in https://openid.net/specs/openid-connect-core-1_0.html#ClaimsParameter is used. With one addition, beside *userinfo* and *id_token* we have added *introspection*.

resources

This are the resource servers and other entities that should be accepted as users of issued access tokens.

issued_at

When the grant was created. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

not_before

If the usage of the grant should be delay, this is when it can start being used. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

expires_at

When the grant expires. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

revoked

If the grant has been revoked.

issued_token

Tokens that has been issued based on this grant. There is no limitation as to which tokens can be issued. Though presently we only have:

- authorization_code,
- access_token and
- refresh_token

id

The grant identifier.

10.2.5 Token

As mention above there are presently only 3 token types that are defined:

- authorization_code,
- access_token and
- refresh_token

A token is described as follows:

```
{
    "type": "authorization_code",
    "issued_at": 1605452123,
    "not_before": 0,
    "expires_at": 1605452423,
    "revoked": false,
    "value":
↪"Z0FBQUFBQmZzVUZieDFWZy1fbjE2ckxvZkFTVC1ZTHJIVlk0Z09tOVk1M0RsOVNDbkdfLTIxTUhILWs4T29kM1lmV015UEN1UGxr
↪",
    "usage_rules": {
        "expires_in": 300,
        "supports_minting": [
            "access_token",
            "refresh_token",
            "id_token"
        ],
        "max_usage": 1
        },
```

(continues on next page)

(continued from previous page)

```
    "used": 0,
    "based_on": null,
    "id": "96d19bea275211eba43bacde48001122"
}
```

type

The type of token.

issued_at

When the token was created. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

not_before

If the start of the usage of the token is to be delay, this is until when. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

expires_at

When the token expires. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

revoked

If the token has been revoked.

value

This is the value that appears in OIDC protocol exchanges.

usage_rules

Rules as to how this token can be used:

expires_in Used to calculate expires_at

supports_minting The tokens types that can be minted based on this token. Typically a code can be used to mint ID tokens and access and refresh tokens.

max_usage How many times this token can be used (being used is presently defined as used to mint other tokens). An authorization_code token can according to the OIDC standard only be used once but then to, in the same session, mint more then one token.

used

How many times the token has been used

based_on

Reference to the token that was used to mint this token. Might be empty if the token was minted based on the grant it belongs to.

id

Token identifier

10.3 Session Info API

10.3.1 add_subordinate

…

10.3.2 remove_subordinate

…

10.3.3 revoke

…

10.3.4 is_revoked

…

10.3.5 to_json

…

10.3.6 from_json

…

10.4 Grant API

...

10.5 Token API

...

10.6 Session Manager API

...

10.6.1 create_session

Creating a new session is done by running the create_session method of the class SessionManager. The create_session methods takes the following arguments.

authn_event An AuthnEvent class instance that describes the authentication event.

auth_req The Authentication request

client_id The client Identifier

user_id The user identifier

sector_identifier A possible sector identifier to be used when constructing a pairwise subject identifier

sub_type The type of subject identifier that should be constructed. It can either be *pairwise* or *public*.

So a typical command would look like this:

```
authn_event = create_authn_event(self.user_id)
session_manager.create_session(authn_event=authn_event, auth_req=auth_req,
                               user_id=self.user_id, client_id=client_id,
                               sub_type=sub_type, sector_identifier=sector_identifier)
```

10.6.2 add_grant

add_grant(self, user_id, client_id, **kwargs)

10.6.3 find_token

find_token(self, session_id, token_value)

10.6.4 get_authentication_event

get_authentication_event(self, session_id)

10.6.5 get_client_session_info

get_client_session_info(self, session_id)

10.6.6 get_grant_by_response_type

get_grant_by_response_type(self, user_id, client_id)

10.6.7 get_session_info

get_session_info(self, session_id)

10.6.8 get_session_info_by_token

get_session_info_by_token(self, token_value)

10.6.9 get_sids_by_user_id

get_sids_by_user_id(self, user_id)

10.6.10 get_user_info

get_user_info(self, uid)

10.6.11 grants

grants(self, session_id)

10.6.12 revoke_client_session

revoke_client_session(self, session_id)

10.6.13 revoke_grant

revoke_grant(self, session_id)

10.6.14 revoke_token

revoke_token(self, session_id, token_value, recursive=False)

CHAPTER

ELEVEN

TESTS

```
pip install -r requirements-dev.txt
pytest --cov=oidcop tests/
```

CHAPTER

TWELVE

THE CLIENTS DATABASE

Information kept about clients in the client database are to begin with the client metadata as defined in https://openid.net/specs/openid-connect-registration-1_0.html .

To that we have the following additions specified in OIDC extensions.

- **https://openid.net/specs/openid-connect-rpinitiated-1_0.html**
  - post_logout_redirect_uri
- **https://openid.net/specs/openid-connect-frontchannel-1_0.html**
  - frontchannel_logout_uri
  - frontchannel_logout_session_required
- **https://openid.net/specs/openid-connect-backchannel-1_0.html#Backchannel**
  - backchannel_logout_uri
  - backchannel_logout_session_required
- **https://openid.net/specs/openid-connect-federation-1_0.html#rfc.section.3.1**
  - client_registration_types
  - organization_name
  - signed_jwks_uri

And finally we add a number of parameters that are OidcOP specific. These are described in this document.

12.1 allowed_scopes

Which scopes that can be returned to a client. This is used to filter the set of scopes a user can authorize release of.

12.2 token_usage_rules

There are usage rules for tokens. Rules are set per token type (the basic set is authorization_code, refresh_token, access_token and id_token). The possible rules are:

- how many times they can be used
- if other tokens can be minted based on this token
- how fast they expire

A typical example (this is the default) would be:

```
"token_usage_rules": {
    "authorization_code": {
        "max_usage": 1
        "supports_minting": ["access_token", "refresh_token"],
        "expires_in": 600,
    },
    "refresh_token": {
        "supports_minting": ["access_token"],
        "expires_in": -1
    },
}
```

This then means that access_tokens can be used any number of times, can not be used to mint other tokens and will expire after 300 seconds which is the default for any token. An authorization_code can only used once and it can be used to mint access_tokens and refresh_tokens. Note that normally an authorization_code is used to mint an access_token and a refresh_token at the same time. Such a dual minting is counted as one usage. And lastly an refresh_token can be used to mint access_tokens any number of times. An *expires_in* of -1 means that the token will never expire.

If token_usage_rules are defined in the client metadata then it will be used whenever a token is minted unless circumstances makes the OP modify the rules.

Also this does not mean that what is valid for a token can not be changed during run time.

CHAPTER

THIRTEEN

FAQ

-