Course by
Consortium GARR
and
Reti
Business & IT Consulting

OIDC authentication flows

OIDC primer - a course on OpenID Connect

Credits: Roland Hedberg, Ioannis Kakavas

Introduction to OpenID Connect

OIDC defines an interoperable way to perform **user authentication**.

1. Clients can **verify the identity** of the end-user based on the authentication performed by an OpenID Provider (acting as an authorization server);
2. It allows clients to **obtain basic profile information** about the end-user in an interoperable and REST-like manner.

Actors involved:

1. The **User** is someone trying to access a protected resource.
2. The **Relying Party** (or Client) is the entity that requests, receives and uses tokens. The RP can be any of a web application, a native application or mobile application.
3. The **OpenID Provider** is the entity that releases tokens. The OP is usually a web based server that is able to receive and process requests for tokens from RPs.

Authentication flows

OpenID supports three flows to authenticate a user and retrieve ID token:

1. **Authorisation code flow** — the most commonly used flow, intended for traditional web apps as well as native/mobile apps. This flow offers optimal security, as tokens are not revealed to the browser and the client app can also be authenticated.
2. **Implicit flow** — for browser (JavaScript) based apps that don't have a backend. The ID token is received directly with the redirection response from the OP. No back-channel request is required here.
3. **Hybrid flow** — rarely used, allows the app front-end and back-end to receive tokens separately from one another. Essentially a combination of the code and implicit flows *(not shown in the following)*.

OpenID endpoints

The endpoints defined in the standard are:

- **Authorize endpoint**: this endpoint performs authentication and authorisation.
- **Token endpoint**: this endpoint allows the requester to get his tokens. If the authorize endpoint is human interaction, this endpoint is machine to machine interaction.
- **UserInfo endpoint**: this endpoint allows you to make a request using your access token to receive claims about the authenticated end-user

Optional endpoints are:

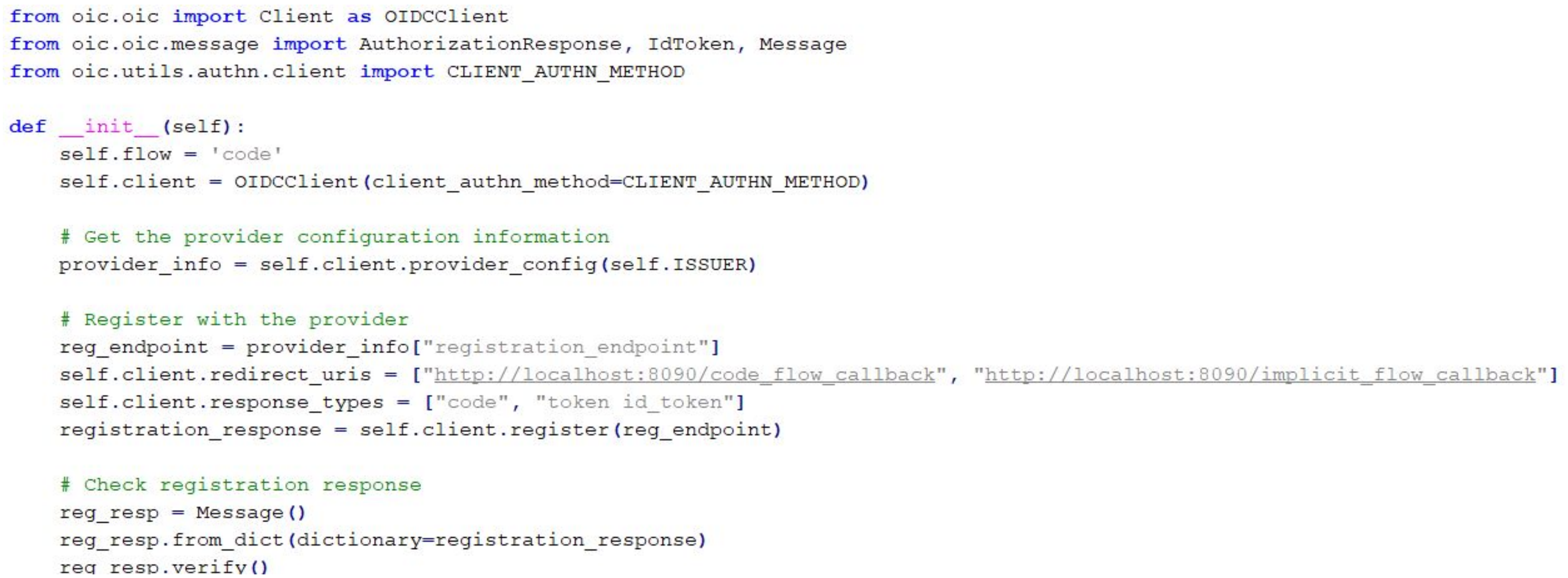
- **Discovery**: this endpoint provide metadata about the OpenID Connect provider, allowing applications to automatically configure for that provider.
- **Client Registration**: this endpoint allow a relying party to register with the OpenID provider.

OpenID authentication summary

In order to use the OpenID connect authentication:

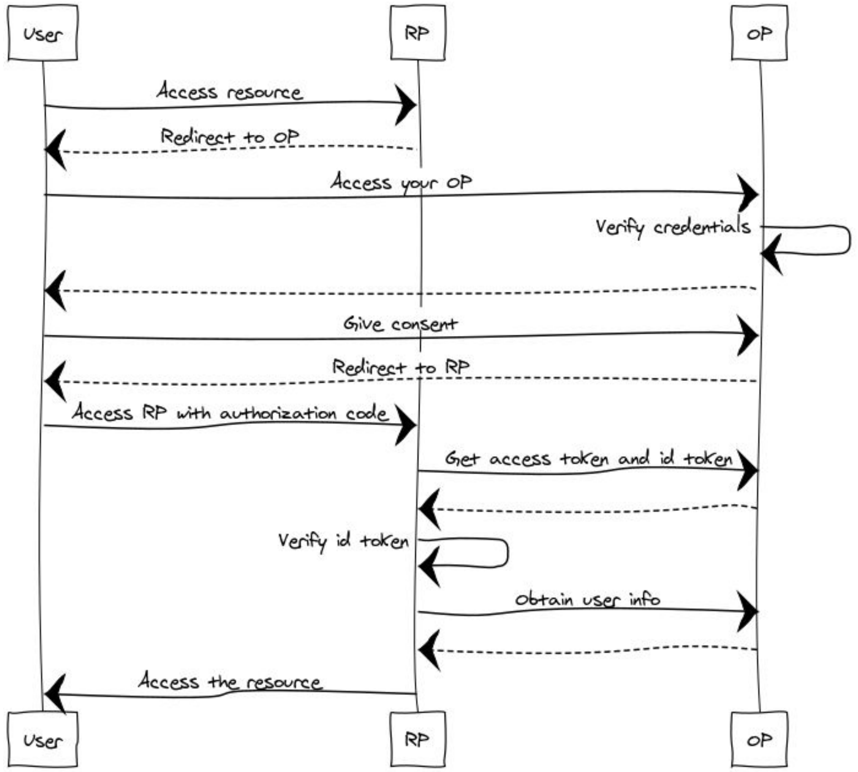
- register a Client (this can happen dynamically or statically) and obtain a client_id & client_secret
- issue an authentication request to the OP endpoint by redirecting the user browser
- the OP will authenticate user (via username/password or any other mechanism)
- the OP will then redirect the user browser to the Client redirection endpoint providing an access token
- [request an ID token to the Token Endpoint of the OP using the access token]
- (optionally) use the access token to retrieve user information

Client Registration - Python Example



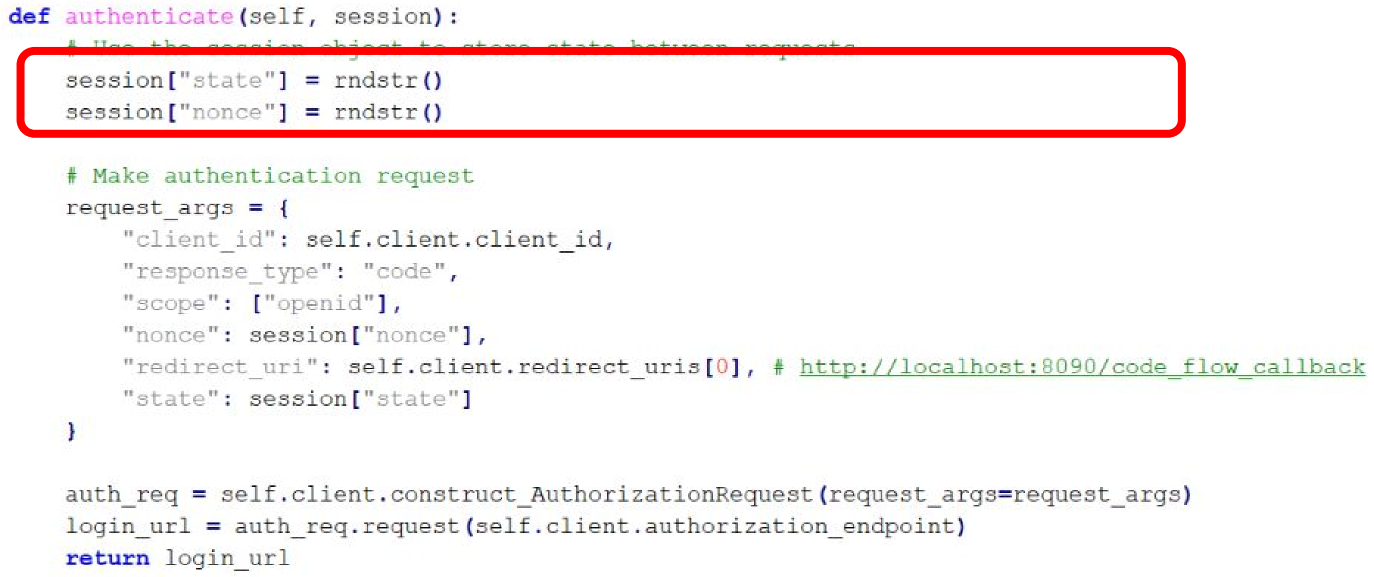
```
from oic.oic import Client as OIDCClient
from oic.oic.message import AuthorizationResponse, IdToken, Message
from oic.utils.authn.client import CLIENT_AUTHN_METHOD

def __init__(self):
    self.flow = 'code'
    self.client = OIDCClient(client_authn_method=CLIENT_AUTHN_METHOD)

    # Get the provider configuration information
    provider_info = self.client.provider_config(self.ISSUER)

    # Register with the provider
    reg_endpoint = provider_info["registration_endpoint"]
    self.client.redirect_uris = ["http://localhost:8090/code_flow_callback", "http://localhost:8090/implicit_flow_callback"]
    self.client.response_types = ["code", "token id_token"]
    registration_response = self.client.register(reg_endpoint)

    # Check registration response
    reg_resp = Message()
    reg_resp.from_dict(dictionary=registration_response)
    reg resp.verify()
```

Authorization code flow



1. User access resource on RP.
2. The RP redirect the user to the OP for authentication.
3. Client sends an Authentication Request containing the desired request parameters to the OP.
4. OP Server Authenticates the End-User by checking credentials.
5. Authorization Server obtains End-User Consent/Authorization.
6. Authorization Server sends the End-User back to the Client with an Authorization Code.
7. Client requests a response using the Authorization Code at the Token Endpoint.
8. Client receives a response that contains an ID Token and Access Token in the response body.
9. Client validates the ID token and retrieves the End-User's Subject Identifier.

Authorization code flow - Python, Authentication Request



```
def authenticate(self, session):
    # Use the session object to store state between requests
    session["state"] = rndstr()
    session["nonce"] = rndstr()

    # Make authentication request
    request_args = {
        "client_id": self.client.client_id,
        "response_type": "code",
        "scope": ["openid"],
        "nonce": session["nonce"],
        "redirect_uri": self.client.redirect_uris[0], # http://localhost:8090/code_flow_callback
        "state": session["state"]
    }

    auth_req = self.client.construct_AuthorizationRequest(request_args=request_args)
    login_url = auth_req.request(self.client.authorization_endpoint)
    return login_url
```

Authorization code flow - State vs Nonce

IN COMMON

- sent to the OP by the Client
- prevent MITM/HIJACKING attacks
- opaque, random (good entropy) strings

DIFFERENT

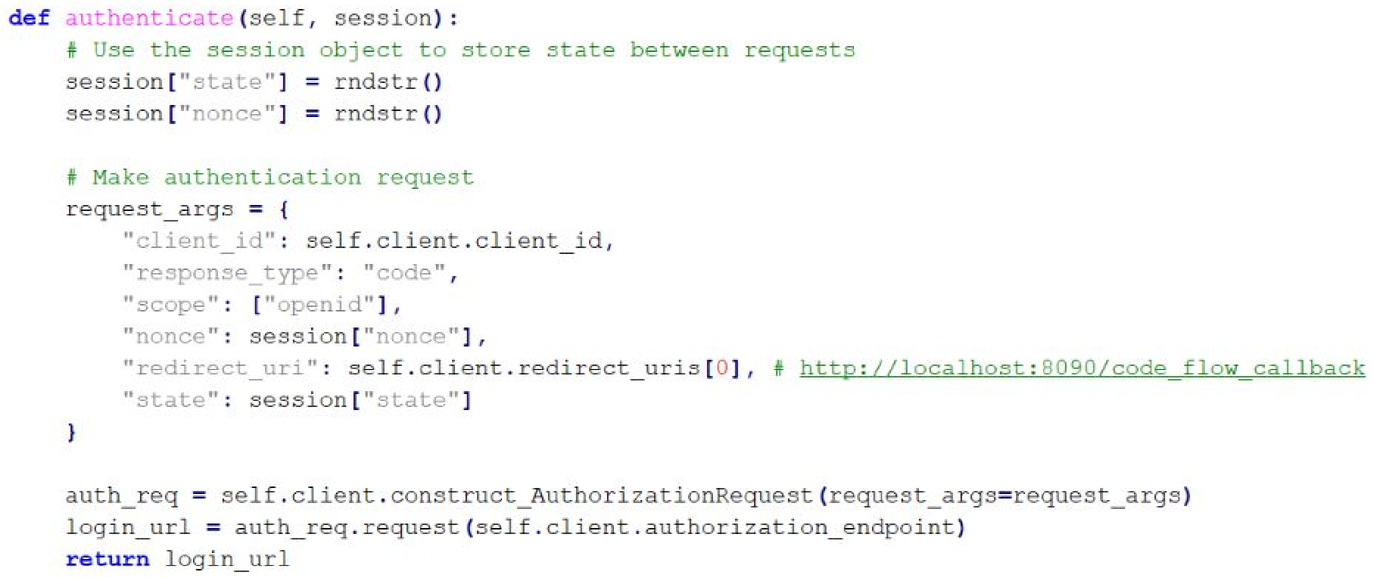
- **state** is used to correlate the authentication response
- **nonce** is used to correlate the identity token coming back

state: Opaque value used to maintain state between the request and the callback. Typically, Cross-Site Request Forgery (CSRF, XSRF) mitigation is done by cryptographically binding the value of this parameter with a browser cookie.

nonce: String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authentication Request to the ID Token. Sufficient entropy MUST be present in the nonce values used to prevent attackers from guessing values.[1]

[1] OpenID Connect Core 1.0 incorporating errata set 1

Authorization code flow - Python, Authentication Request



```python
def authenticate(self, session):
    # Use the session object to store state between requests
    session["state"] = rndstr()
    session["nonce"] = rndstr()

    # Make authentication request
    request_args = {
        "client_id": self.client.client_id,
        "response_type": "code",
        "scope": ["openid"],
        "nonce": session["nonce"],
        "redirect_uri": self.client.redirect_uris[0], # http://localhost:8090/code_flow_callback
        "state": session["state"]
    }

    auth_req = self.client.construct_AuthorizationRequest(request_args=request_args)
    login_url = auth_req.request(self.client.authorization_endpoint)
    return login_url
```

Authorization code flow - Authentication Request

The Authentication Request

```
GET
https://mitreid.org/authorize?nonce=hdq7pY8UeNgvcIhK&state=A0XEujuRJ7vOBiCi&redirect_
uri=http%3A%2F%2Flocalhost%3A8090%2Fcode_flow_callback&response_type=code&client_id=4
7bbc19f-5bb0-4a68-92fa-920659c542b8&scope=openid+profile+email HTTP/1.1
```

Authentication Request Parameters

```
nonce: hdq7pY8UeNgvcIhK
state: A0XEujuRJ7vOBiCi
redirect_uri: http://localhost:8090/code_flow_callback
response_type: code
client_id: 47bbc19f-5bb0-4a68-92fa-920659c542b8
scope: openid profile+email
```

Authorization code flow - Python, callback

```python
def code_flow_callback(self, auth_response, session):
    # Parse the authentication response
    aresp = self.client.parse_response(AuthorizationResponse, info=auth_response, sformat="urlencoded")
    assert aresp["state"] == session["state"]

    # Make token request
    access_code = aresp["code"]
    args = {
        "code": access_code,
        "redirect_uri": self._get_redirect_uris_for_auth(),
        "client_id": self.client.client_id,
        "client_secret": self.client.client_secret
    }

    resp = self.client.do_access_token_request(scope=aresp["scope"],
                                               state=aresp["state"],
                                               request_args=args,
                                               authn_method="client_secret_post")

    # Validate the ID Token according to the OpenID Connect spec (sec 3.1.3.7.)
    id_token_claims = IdToken()
    id_token_claims.from_dict(dictionary=resp['id_token'])
    id_token_claims.verify()

    # Make userinfo request
    userinfo = self.client.do_user_info_request(state=aresp["state"])

    # Set the appropriate values
    access_token = resp['access_token']
    return success_page(access_code, access_token, id_token_claims, userinfo)
```

Authorization code flow - Authentication Response

The Authentication Response (via User-Agent Redirection)

```
GET http://localhost:8090/code_flow_callback?code=N9SPuU&state=A0XEujuRJ7vOBiCi
HTTP/1.1
```

Authentication Response Parameters

```
code: N9SPuU
state: A0XEujuRJ7vOBiCi
```

Authorization code flow - Token Request

```python
def code_flow_callback(self, auth_response, session):
    # Parse the authentication response
    aresp = self.client.parse_response(AuthorizationResponse, info=auth_response, sformat="urlencoded")
    assert aresp["state"] == session["state"]

    # Make token request
    access_code = aresp["code"]
    args = {
        "code": access_code,
        "redirect_uri": self._get_redirect_uris_for_auth(),
        "client_id": self.client.client_id,
        "client_secret": self.client.client_secret
    }

    resp = self.client.do_access_token_request(scope=aresp["scope"],
                                               state=aresp["state"],
                                               request_args=args,
                                               authn_method="client_secret_post")

    # Validate the ID Token according to the OpenID Connect spec (sec 3.1.3.7.)
    id_token_claims = IdToken()
    id_token_claims.from_dict(dictionary=resp['id_token'])
    id_token_claims.verify()

    # Make userinfo request
    userinfo = self.client.do_user_info_request(state=aresp["state"])

    # Set the appropriate values
    access_token = resp['access_token']
    return success_page(access_code, access_token, id_token_claims, userinfo)
```

Authorization code flow - Token Request

Token Request

```
POST https://mitreid.org/token

Body:
code=<REDACTED>
&state=A0XEujuRJ7vOBiCi
&redirect_uri=http%3A%2F%2Flocalhost%3A8090%2Fcode_flow_callback
&client_id=927305c2-cca2-4686-8134-5e9ed0e76149
&client_secret=<REDACTED>
&grant_type=authorization_code
```

Authorization code flow - Token Response

Token Response

```
HTTP/1.1 200 OK
Content-Type: application/json
[..]

{
      "access_token":"<REDACTED>",
      "token_type":"Bearer",
      "Expires_in":3599,
      "scope":"openid email profile",
      "id_token":"eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIwMTkyMS5GTEFOUkpRVyIsImF1ZCI6Ijk
      yNzMwNWMyLWNjYTItNDY4Ni04MTM0LTVlOWVkMGU3NjE0OSIsImtpZCI6InJzYTEiLCJpc3MiOiJodHRwczpcL1wvbWl0c
      mVpZC5vcmdcLyIsImV4cCI6MTQ5ODQ4MTQ5NiwiaWF0IjoxNDk4NDgwODk2LCJub25jZSI6IkVNTkRDb1JJTnhOZWFaMlg
      iLCJqdGkiOiJkYTk4MTIzYi0wYjk3LTQ2NjAtOTU1NC1iNTQxNGZiM2VjZGEifQ.f8FWz12XIRqfiGt_LdnCb4-rOsBDgx
      V3AmVqvijPKMIl-Yt2D2kdKREGhZyUo-tvv1xcU0NDZXkFrLSfOFYY6et0gtbFvVkLuOiZi4LqytHqUWPP-dfXiuNxRDCN
      eQ5lQydESZmeVSvrNOIoZ34PdwyfU_MWMWLojf9r6r68zdYun3eSwykSIixfrJqmRh2n1Rl3RagKrx_CzB_LOR-ALa1hmt
      1TCnOheuVjzRipbfsa34d7NM0Cd0kJKC5v5FOytC2VbGFTpzlRPcLFR2euxryEF1TjquxonA3eR_QqQ-yKvIV6_RrZ2DSr
      Ls5voedRXfoBkwEliKYgRMQLvtII9Q"
}
```

Header

Payload

Signature

Access Token response payload dissection

CLI

```
$ base64 -d << EOF
eyJzdWIiOiIwMTkyMS5GTEFOUkpRVyIsImF1ZCI6IjkyNzMwNWMyLWNjYTItNDY4Ni04MTM0LTVlOWVkMGU3NjE0OSIsImtpZCI6InJz
YTEiLCJpc3MiOiJodHRwczpcL1wvbWl0cmVpZC5vcmdcLyIsImV4cCI6MTQ5ODQ4MTQ5NiwiaWF0IjoxNDk4NDgwODk2LCJub25jZSI6
IkVNTkRDb1JJTnhOZWFaMlgiLCJqdGkiOiJkYTk4MTIzYi0wYjk3LTQ2NjAtOTU1NC1iNTQxNGZiM2VjZGEifQ
EOF
{"sub":"01921.FLANRJQW","aud":"927305c2-cca2-4686-8134-5e9ed0e76149","kid":"rsa1","iss":"https:\/\/mitre
id.org\/","exp":1498481496,"iat":1498480896,"nonce":"EMNDCoRINxNeaZ2X","jti":"da98123b-0b97-4660-9554-b5
414fb3ecda"}
```

… nicely formatted…

```
{
      "sub":"01921.FLANRJQW",
      "aud":"927305c2-cca2-4686-8134-5e9ed0e76149",
      "kid":"rsa1",
      "iss":"https:\/\/mitreid.org\/",
      "exp":1498481496,
      "iat":1498480896,
      "nonce":"EMNDCoRINxNeaZ2X",
      "jti":"da98123b-0b97-4660-9554-b5414fb3ecda"
}
```

Authorization code flow - UserInfo Request

```python
def code_flow_callback(self, auth_response, session):
    # Parse the authentication response
    aresp = self.client.parse_response(AuthorizationResponse, info=auth_response, sformat="urlencoded")
    assert aresp["state"] == session["state"]

    # Make token request
    access_code = aresp["code"]
    args = {
        "code": access_code,
        "redirect_uri": self._get_redirect_uris_for_auth(),
        "client_id": self.client.client_id,
        "client_secret": self.client.client_secret
    }

    resp = self.client.do_access_token_request(scope=aresp["scope"],
                                               state=aresp["state"],
                                               request_args=args,
                                               authn_method="client_secret_post")

    # Validate the ID Token according to the OpenID Connect spec (sec 3.1.3.7.)
    id_token_claims = IdToken()
    id_token_claims.from_dict(dictionary=resp['id_token'])
    id_token_claims.verify()

    # Make userinfo request
    userinfo = self.client.do_user_info_request(state=aresp["state"])

    # Set the appropriate values
    access_token = resp['access_token']
    return success_page(access_code, access_token, id_token_claims, userinfo)
```

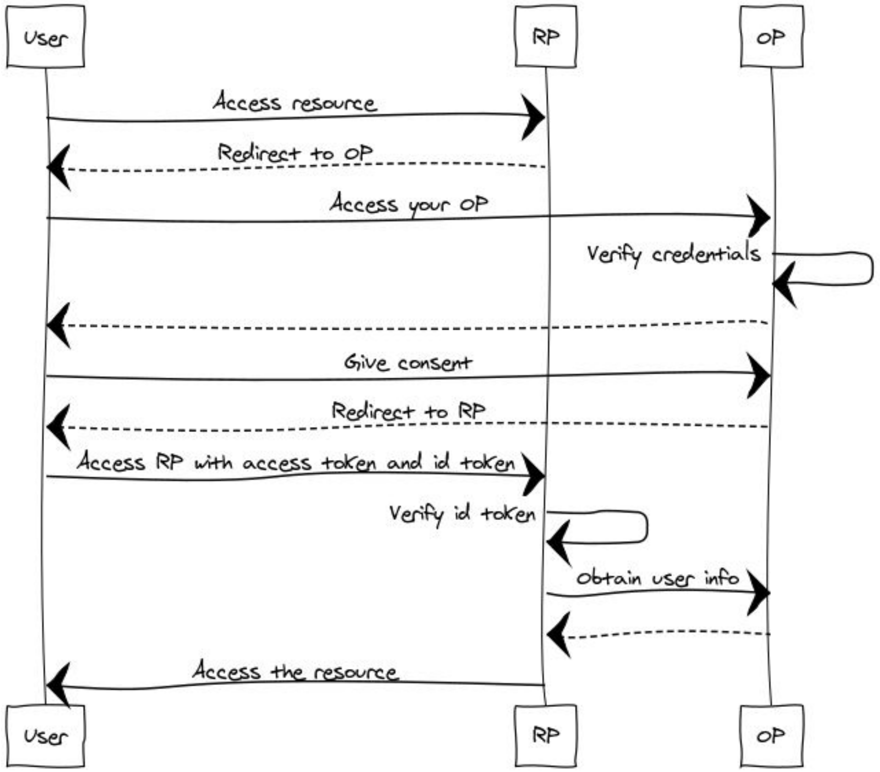
UserInfo Request

```
GET https://mitreid.org/userinfo HTTP/1.1

Authorization: Bearer eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJ1c2VyIiwiYXpwIjoiZTk5ZGVhNT
MtOTM4Yi00MzMxLTk4ODQtZWI1OTVkZjI1M2ExIiwiaXNzIjoiaHR0cHM6XC9cL21pdHJlaWQub3JnXC8iLCJleHAiOjE0OTg0O
Dk3MjEsImlhdCI6MTQ5ODQ4NjEyMSwianRpIjoiYzM4ZDUyODUtNGE4MS00MjNlLWFlNjEtNjQ2NDI5OGU3OGM5In0.XLPb4a5K
t6u7p5rUmVA2I6zFrRwqbJdSAZRQ861Y0jltO4dBmCpXQatoPYCHlY_Kwc3sBjlLGg7ibo4HPfnLgosY-y3iq-DrI-kCnMrIs5m
d8cRvj-DLGuaeucvRP95qLAI2HaICSUKfa2IG-Oi1gnP3P2mXPnVWsBXp5QHTIxoyUS3_jOUb6tKR_rWMVVMtGb-Jgh37PWWQB2
1dBRSjHp0W5MzGq8_7pV5Lm-s8d7gvnA_S4WBmRfb9u2ZJvicXHSr_GtdhWiGMfVSAZlrZVm7GmyRWvowegfQuLpTo1lADPofBL
4a7pBG5PAH96FbwT0BiDTkju3pu1Ibg8axegQ
```

UserInfo Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{
      "sub":"01921.FLANRJQW",
      "name":"Demo User",
      "preferred_username":"user",
      "email":"user@example.com",
      "email_verified":true
}
```

Implicit flow



1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an ID Token and, if requested, an Access Token.
6. Client validates the ID token and retrieves the End-User's Subject Identifier.

Implicit flow - Python

```python
def authenticate(self, session):
    # Use the session object to store state between requests
    session["state"] = rndstr()
    session["nonce"] = rndstr()

    # Make authentication request
    request_args = {
        "client_id": self.client.client_id,
        "response_type": ["id_token", "token"],
        "scope": ["openid"],
        "nonce": session["nonce"],
        "redirect_uri": self.client.redirect_uris[1], # http://localhost:8090/implicit_flow_callback
        "state": session["state"]
    }

    auth_req = self.client.construct_AuthorizationRequest(request_args=request_args)
    login_url = auth_req.request(self.client.authorization_endpoint)
    return login_url
```

Implicit flow - Python, callback

```
def implicit_flow_callback(self, auth_response, session):
    # Parse the authentication response
    aresp = self.client.parse_response(AuthorizationResponse, info=auth_response, sformat="urlencoded")
    assert aresp["state"] == session["state"]

    # Validate the ID Token according to the OpenID Connect spec (sec 3.2.2.11.)
    id_token_claims = IdToken()
    id_token_claims.from_dict(dictionary=aresp['id_token'])
    id_token_claims.verify()

    # Make userinfo request
    userinfo = self.client.do_user_info_request(state=aresp["state"])

    # Set the appropriate values
    access_code = None
    access_token = aresp['access_token']

    return success_page(access_code, access_token, id_token_claims, userinfo)
```

Q&A

Thanks for your attention!