

[FOR SWD] HOW TO CALL API USING OAUTH AUTHORIZATION CODE FLOW

Version	Changes	Remarks
0.1	Initial Version	
0.2	Removed DPoP requirement, updated OAuth code flow, requirement for JWKS endpoint (EC curve) and updated sample codes.	
0.3	Updated expiry time to 5 minutes as permitted by NDI. Updated with missing code_verifier in Step 8.	

A. Overview

This guide explains how to implement the OAuth 2.1 Authorization Code Flow to allow a CorpPass user to grant a third-party client application access to the Corppass user's protected resources.

Learning Outcomes

- Understand OAuth 2.1 Authorization Code Flow
- Set up your Client with OAuth 2.1 Authorization Code Flow

Terminologies Used

Term	Definition
Client	The application that is attempting to act on the user's behalf to access the user's resources.
Authorization Server	The authorization server is what the user interacts with when a client is requesting access to their account. This server displays the consent prompt for user to approve or deny the request. The authorization server is also responsible for granting access tokens after the users authorizes the application.
Authorization Code	A temporary code that the Authorization Server generates when the user authorizes a request. The client will exchange the authorization code for an access token.
Access Token	The thing that allows a client to access an API. It is a string that represents authorization granted to the client by the user. It represents specific scopes and duration of access. It is granted by the user and enforced by the authorization server.
Resource Server	The resource server (aka API server) is the server that hosts the protected resources and uses the access tokens to accept and respond to protected resource requests.
SWD	Software Developer (you) developing the client.
Code Verifier	Randomly generated string
Code Challenge	Hashed value of Code Verifier
APEX	API Exchange Portal that developer can discover APIs, learn how to use them, request access and try them out.

B. Prerequisites (Initial Setup)

1. Create Public and Private Keys for the JSON Web Key Set (JWKS)

The JWKS is a set of keys containing the public keys used to verify any JSON Web Token (JWT). Generate a set of asymmetric private and public keys for signing of claims for connecting to Authorization Server using EC (P-256). An example of an online key generator is at <https://mkjwk.org>, and sample code is as below. Save the private key. You will need it for JWT signing in later steps.

Sample Code to Generate JWT

```
// DEPENDENCIES
const jose = require('node-jose');
/**
 * @module crypto
 * @description Node.js built-in crypto module
 * @link https://nodejs.org/api/crypto.html#crypto_crypto
 */
const crypto = require("crypto");
/**
 * @module crypto
 * @description JSON Object Signing and Encryption (JOSE) module via NPM
 * @link https://www.npmjs.com/package/node-jose
 */
const jose = require('node-jose');

// CREATE KEYSET
async function key(){
  try{
    let key = crypto.generateKeyPairSync('ec', {
      namedCurve: 'prime256v1',
      publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
      },
      privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem'
      }
    })
    let cryptoKey = await jose.JWK.asKey(key.privateKey, "pem");
    console.log(cryptoKey.toPEM())           // X.509 public key
    console.log(cryptoKey.toPEM(true))       // X.509 private key
    console.log(cryptoKey.toJSON())           // JSON format of public key
                                           // This can be used for the JWKS endpoint
    console.log(cryptoKey.toJSON(true))       // JSON format of private key

  } catch(e) {console.log(e);}}

key();
```

2. Provide your JWKS Endpoint

Host the JWKS generated in the previous step on a publicly accessible URL. You will use the key-id ('kid') in later requests. Information on JWK Set format and JWK formats can be found in [RFC7517](https://tools.ietf.org/html/rfc7517). The keys 'kid' and 'use' may have to be manually added.

Sample JWKS Endpoint (eg. https://{SWD_domain}/.well-known/keys.json)

```
{
  "keys": [
    {
      kty: "EC",
      kid: "key-id1",
      crv: "P-256",
      alg: "ES256",
      use: "sig",
      x: "SXUBSK5y9HO2Sjrac-N9VbSQaLAtePjtWGab9_Yoo1c",
      y: "1-aB05rylpiFcTNx-3H9JFUXtxoWXmLWOWO_5eqGgw4"
    }
  ]
}
```

3. Create an App in the Developer Console

1. Sign into the API portal and click on **Applications** on the top bar. Alternatively, you may [sign up for a developer account](#).
2. Click on **Create Application**.
3. Provide a unique name for your application.
4. Click on **Save Application**.

4. Configure the App

Before the client credentials can be used, some additional configuration is required.

Redirect URI

During the OAuth flow, users are redirected to their browser to authenticate and then authorize the app to transact on their behalf. Prior to redirecting the user, the Authorization Server verifies the `redirect_uri` parameter passed into the authorization URL matches one of the redirect URIs configured for the application. This is an exact match check, i.e. both URIs must be the same.

You can configure the redirect URIs under the OAuth Redirect URI section of the App.

Redirect URIs must begin with `https`.

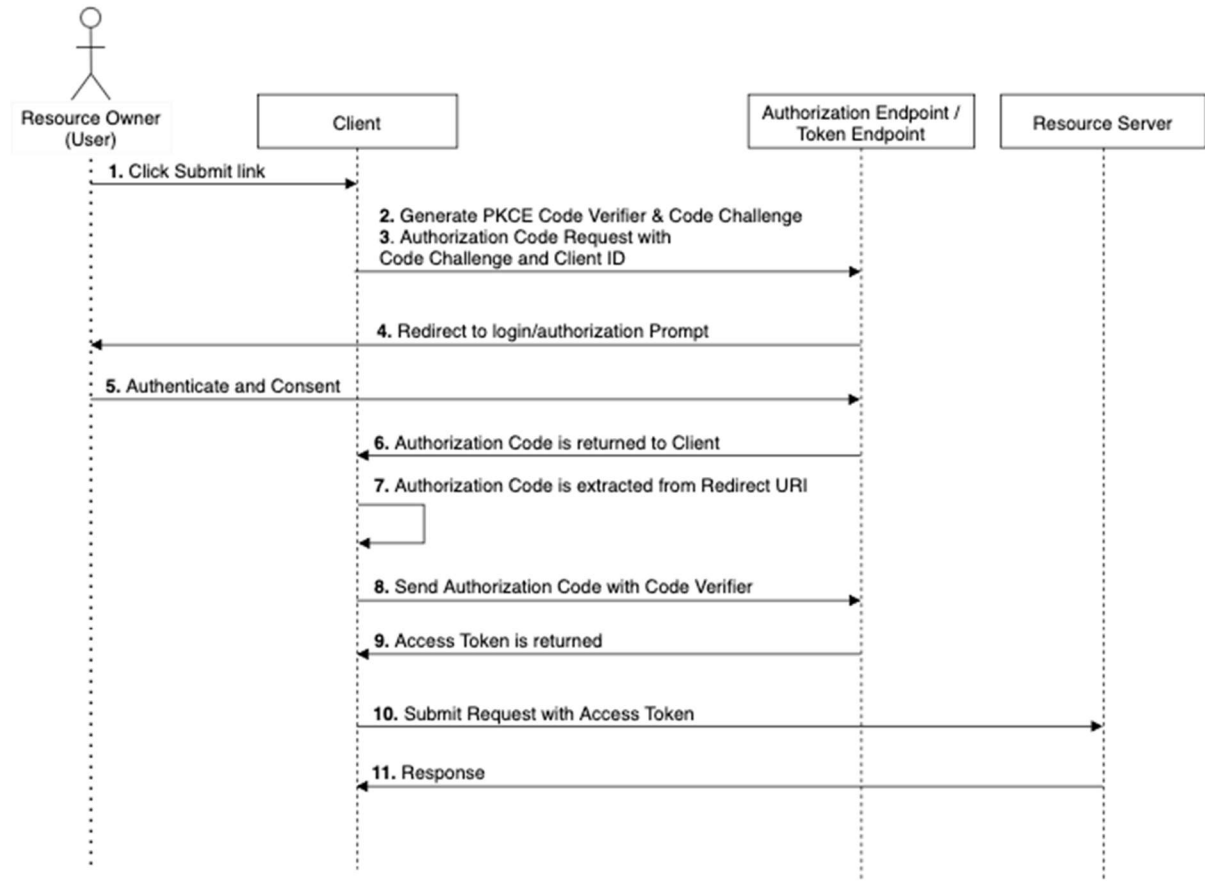
1. Sign into the API portal and click on **Applications** on the top bar.
2. Select your application from the list.
3. Click on **Edit Application**. By default, you are in the **Details** tab.
4. Navigate to the **Authentication** tab.
5. Expand the **OAuth Credentials** section.
6. Click on **Generate** to generate a new set of OAuth credentials.
7. Enter a redirect URL.
8. Click on **Generate Client**.

JWKS Endpoint

Provide the JWKS endpoint created in the previous step. Click **Save**.

C. How to call APIs using the OAuth Authorization Code Flow

OAuth 2.1 Authorization Code Flow (with PKCE)



Staging and Production URLs

Endpoint	Access Mechanism	Staging / Production
Authorization Endpoint	Public TLS	<u>Staging</u> https://public-stg.api.gov.sg/oauth/cp/v2/authorize <u>Production</u> https://public.api.gov.sg/oauth/cp/v2/authorize
Token Endpoint	Public TLS w/Client Assertions	<u>Staging</u> https://public-stg.api.gov.sg/oauth/cp/v2/token <u>Production</u> https://public.api.gov.sg/oauth/cp/v2/token
Resource Server	Public TLS	<u>Staging</u> <a href="https://public-stg.api.gov.sg/<agency>/<api>">https://public-stg.api.gov.sg/<agency>/<api> <u>Production</u> <a href="https://public.api.gov.sg/<agency>/<api>">https://public.api.gov.sg/<agency>/<api>

1. User Clicks Sign in/Authorize/Submit

In your client application, you want to provide a button to sign the user in. When the user clicks this button, they are redirected to SingPass sign-in page so they can authenticate and then authorize themselves. The next 2 steps help you build the URL for this request.

2. Client Generates PKCE Code Verifier & Code Challenge

Generate the PKCE Code Verifier and Code Challenge. [Learn more about PKCE here](#).

- Code Verifier: Random URL-safe string with a minimum length of 43 characters
- Code Challenge: Base64 URL-encoded SHA-256 hash of the code verifier.

The generated PKCE code will look like this:

```
code_verifier: owFWUGrbZ3Bk5epaomy2EYkMCQNZnkjoL_H79Gv02u0
code_challenge: 3yj50_1LB91nSs9DyzZ5tPZh5H0NxWLwNtYBX0pOrII
```

The Client sends the *code_challenge* in the authorization request, while the *code_verifier* is sent in the Access Token request later.

A sample code to generate the *code_challenge* and *code_verifier* might look like this:

```
/**
 * @module crypto
 * @description Node.js built-in crypto module
 * @link https://nodejs.org/api/crypto.html#crypto_crypto
 */
const crypto = require("crypto");

// LOCAL FUNCTIONS
const base64URLEncode = function base64URLEncode(str) {
  return str.toString('base64')
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');
}

const sha256 = (buffer) => {
  return crypto.createHash("sha256").update(buffer).digest();
};

const verifier = base64URLEncode(crypto.randomBytes(32));
const challenge = base64URLEncode(sha256(verifier));
```

Source: [Auth0](#)

3. Client Request to the Authorization Endpoint (Authorization Code Request) with the Generated Code Challenge and Client ID

The Client will make a request to the Authorisation Endpoint. The request URL would be like this:

```
https://public.api.gov.sg/oauth/cp/v2/authorize?client_id={client_id}&redirect_uri={redirect_uri}&response_type=code&scope={scope}&code_challenge_method=S256&code_challenge={code_challenge}
```

Parameters are as follows:

- `{redirect_uri}` is the Redirect URI of the Application (URL encoded).
- `{scope}` is the OAuth2.1 scope which is provided by the Resource Server owner.
- `{code_challenge}` is the code challenge used for PKCE (generated in Step 2).
- `{client_id}` is the Client ID generated in Initial Setup.

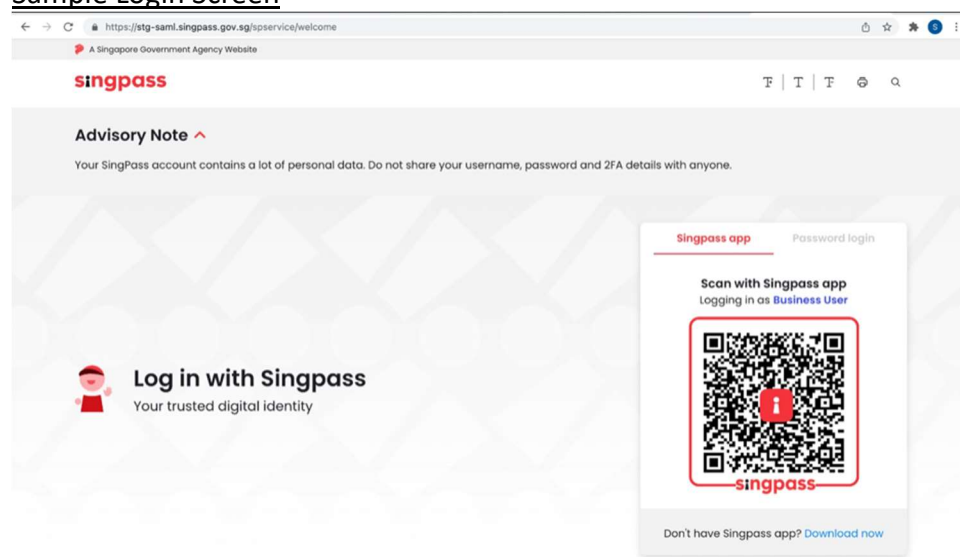
Optional Parameter is as follows:

- `{state}` is the session ID value used by Client to maintain state between request and callback.

4. Client User is Redirected to the Login and Authorization Pages

The login and consent screens are shown to the user.

Sample Login Screen



Sample Consent Screen

Request for Permission

APEX PAYROLL SERVICE CP is requesting permission for the purpose of

Submission of CPF Contributions
Submission of Employment Income Records

Do you agree to connect and transact with **Government Technology Agency (APEX)** based on the [Terms of Use](#)?

YES

NO

5. The User Authenticates and Authorizes the Client at the Login and Consent Pages. The User authenticates and authorizes the client in the Login (Singpass) and Consent screens.

6. The Authorization Server Redirects User Back to the Client with an Authorization Code.

The user is redirected back to the *Redirect URI* of the Application.

A redirected URL with parameters might look like this:

```
https://{redirect_uri}?code=S1q-_k4NUrh5fWLqFz7CwtDp12ntJbJuvMz1CzBK
```

7. Client Extracts Authorization Code from Redirect URI

The client extracts the *Authorization Code* from the *Redirect URI*. In the above example, the code extracted will be:

```
S1q-_k4NUrh5fWLqFz7CwtDp12ntJbJuvMz1CzBK
```

8. Client sends this Authorization Code to the Token Endpoint, with a Code Verifier

Exchange the code for Access Token

To retrieve the Access Token, the Client will pass the Authorization Code to the Token Endpoint along with the *code_verifier* (generated in Step 2).

The specifications for the request to the Token Endpoint are as follows:

Specification	Information	
Token Endpoint	https://public.api.gov.sg/oauth/cp/v2/token	
method	POST	
content-type	'application/x-www-form-urlencoded'	
body	'code'	String of parameter retrieved from Step 7 above
	'client_assertion'	{client_assertion} specified below
	'client_id'	String of Client ID obtained in the Initial Setup
	'client_assertion_type'	'urn:ietf:params:oauth:client-assertion-type:jwt-bearer'
	'grant_type'	'authorization_code'
	'redirect_uri'	String of exact redirect URI string provided in the Initial Setup in URL encoded format
	'code_verifier'	String of PKCE Code Verifier generated in Step 2 above

Client Assertion

Additional Information on *client_assertion* is as follows:

- {client_assertion} is a JSON formatted list of claims based on RFC7523:

Claims	Information
'iss'	Issuer, also equal to <i>Client ID</i> obtained in the Initial Setup
'sub'	Subject, also equal to <i>Client ID</i> obtained in the Initial Setup
'aud'	Audience, also equal to <i>NDI's URL of Token Endpoint</i>
'kid'	Key ID, also equal to the JWT Key ID (public key of the keyset used to sign the <i>client_assertion</i>) in the <i>JWKS Endpoint</i> described above in Initial Setup (Section B1)
'jti'	JWT ID, an arbitrary value to identify the Access Token request, like a session ID
'iat'	Current epoch time of epoch time, in seconds
'exp'	Expiry time in Linux time + 5 minutes, in seconds

A sample code to generate the signed client assertion might look like this:

```
/**
 * @module jsonwebtoken
 * @description JSON Web Tokens module via NPM
 * @link https://www.npmjs.com/package/jsonwebtoken
 */
const jwt = require('jsonwebtoken');
/**
 * @module fs
 * @description Node.js built-in File System module
 * @link
 */
const fs = require('fs');
const epoch = new Date().getTime();

let payload = {};
let privateKey = fs.readFileSync('./private.key', 'utf8'); // your private key generated in Initial
Setup in PEM format
let signOptions = {
  issuer: 'STG-xxxx', // your Client ID issued during Initial Setup
  subject: 'STG-xxxx', // your Client ID issued during Initial Setup
  audience: 'https://public.api.gov.sg/oauth/cp/v2/token',
  jwtid: 'session-' + epoch, // your Session ID - used for troubleshooting and preventing replay
  attacks
  keyid: 'key-id1', // your key ID in JWKS endpoint from Initial Setup
  expiresIn: '5m',
  algorithm: 'ES256'
}

let clientAssertion = jwt.sign(payload, privateKey, signOptions);
// Signing Client Assertion will prove client is the issuer of client assertion
```

9. The Token Endpoint Returns the Access Token

The response (in a format specified in RFC6749) will contain the Access Token.

```
{
  "access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ii9ScQZzZ4d09NdmJ0dDZhalld1WmU2R2xncy1qM3dtNXJpQXlDVW9SYXNhLUkifQ.eyJzdWIiOiIxODAwOTU2MjVhX2FiYjU0TE4LTzJyYWYtNDI4Yi04ZTI1LWVhY2JhZGRiODUyNiIsImp0aSI6IiZPbDMxZ3ZtTwd1S0tydk9JT2RRRGwzNmhrQXRUY09sWjVoSVp0ZkQiLCJzY29wZSI6IkNwZkNvb1N1YiBFbXBjbmNvbWVtdWIIiLCJleHBpcmVzX2luIjoxODAwLCJhdWQiOiJTVGEcyLVRFU1QtQ0xJRUSULU1BTkdFTUVOV0C04dGVzdCIsInJlYWxtIjoieYXBLEC1jcCIsImIzcyI6Imh0dHBzOi8vdGVzdC5hcGkubXlpbmZvLmdvdi5zYy9zZXJ2aWNL1YXV0aC9hcGV4LWwIiwia2xpZW50Ijpw7ImNsawVudF9pZCI6IiNURzItVEVTV0C04dGVzdCIsImVudGl0eV91ZW4iOiJUMTZhQjAwMDJHIiwia2Z5aXN0IjoiJHb3Zlcm5tZW50IFRlY2hub2xvZ3kgQWdlbmN5IChBUeVYKSj9LCJpYXQiojE2NTYzNzgzMtMsIm5iZiI6MTY1NjM3ODMxMywiZXBhIjoxNjU2MzgWMTZfQ. rvySU31cJBgrhd6HfIAdVgy1nEtKbXH64kER_8Lj58v_wAGES0u2KFxSI85L4vkG1vwuRwEGBwKANQJpLe-bT0KY0XPiumGfVbPTs-ImIukBSIPDSkqv49mBPr79kVQFjp-YN53VaVbXcWcdt2JxD70P8w-5ocC0ms8W7lFwCwmBW1x8i7HSDWKrtkPJavmqmTca0DfYmJXUcccw68_s520SAZI1dx3WCfOp-vQ8ogmEEdVwjZ_sG11cApCVfCUGlvvHsk_nSQz_8_SUnE1BFiPa0xmyWxC-6xmVZCYUwsV3t2WDallzOuGWBgkUi1YU_MUZN_7P9CM088e071p9ZQ",
  "token_type": "Bearer",
  "expires_in": 299,
  "scope": "CpfConSub EmpIncomeSub"
}
```

10. Client Submits Request to the Resource Server with the Access Token

The Client submits request to the Resource Server with the Access Token as a Bearer token format in the Authorization header of the HTTP request.

```
curl --request GET \
  --url https://public.api.gov.sg/{agency}/{api} \
  --header 'authorization: Bearer {ACCESS_TOKEN}' \
  --header 'content-type: application/json' \
  --header 'ApiId: {API Key}'
```

11. Resource Server Validates the Access Token Before Returning a Response

Once the access token and the scopes have been validated, the resource server process the request and return a response to the client.

D. Further Reference

- RFC7517 JSON Web Key (JWK) (<https://datatracker.ietf.org/doc/html/rfc7517>)
- RFC6749 OAuth2.0 (<https://datatracker.ietf.org/doc/html/rfc6749>)
- RFC7636 Proof Key for Code Exchange by OAuth Public Clients (<https://datatracker.ietf.org/doc/html/rfc7636>)
- OAuth2.1 (<https://oauth.net/2.1/>)
- RFC7523 JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants (<https://datatracker.ietf.org/doc/html/rfc7523>)
- RFC4492 Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) (<https://datatracker.ietf.org/doc/html/rfc4492>)