databricks

# OAuth Integration Guide for Cloud-Based Partners

Authors : Databricks Technology Partner Team
Last updated : Sep 27, 2024

## Introduction

This document provides detailed instructions for integrating your cloud-based application with Databricks using OAuth. It is specifically tailored for partners using Databricks DBSQL drivers.

## OAuth Overview

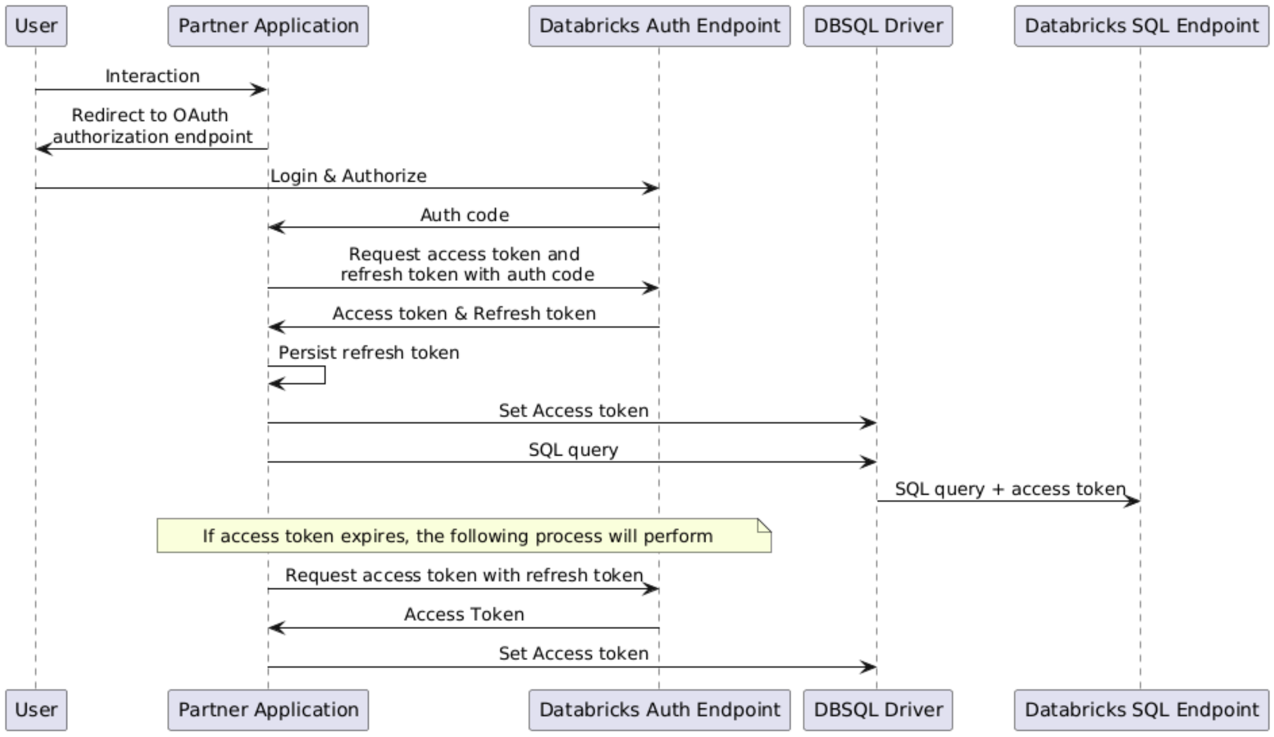
OAuth is a widely accepted industry standard for authorization. This guide focuses on two primary OAuth scenarios: interactive mode or OAuth User-to-Machine (U2M) and Machine-to-Machine (M2M).

### OAuth User-to-Machine (U2M)

It is also known as the 3-legged OAuth and OAuth Authorization Code Flow (defined in OAuth 2.0 RFC 6749, section 4.1). The "three legs" refer to the interactions between the user, the partner application, and the Databricks OAuth endpoint.

- **Use Case**: This flow is suitable for the partner applications that need to access Databricks services on behalf of users. U2M interactions are essential for data scientists, engineers, and analysts who use the Databricks platform for data processing, analysis, and machine learning tasks.

- **Interaction flow**



It is also known as the 3-legged OAuth and OAuth Authorization Code Flow (defined in OAuth 2.0 RFC 6749, section 4.1). The "three legs" refer to the interactions between the user, the partner application, and the Databricks OAuth endpoint.

A typical user-facing OAuth experience in a SaaS application is:

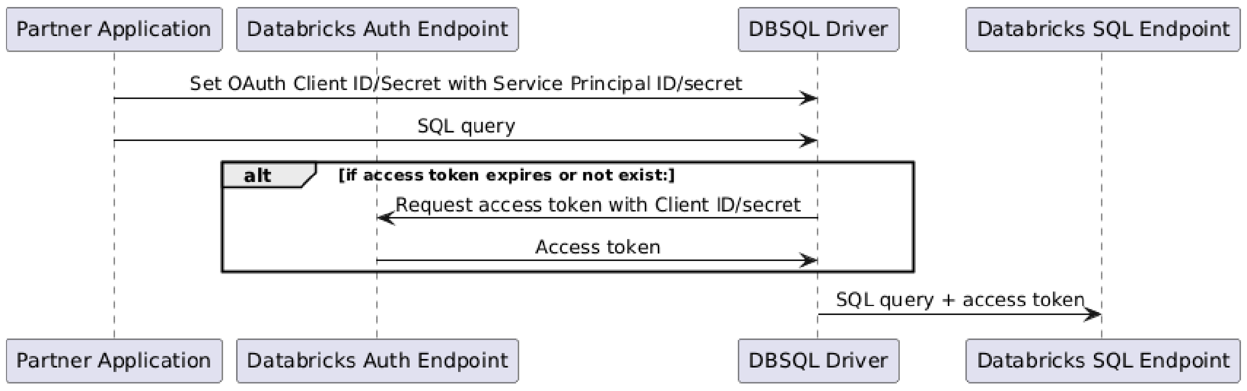
1. A user logs in to the partner's website.
2. The user selects Databricks as the data source, upon connection, the user is redirected to Databricks' login page.
3. After successfully logging in with Databricks, the partner application receives the access token and refresh token from Databricks via the callback registered in step (2).
4. The partner application uses the access token to connect to Databricks SQL or clusters. The partner application should store the refresh token, which will be used to renew the refresh token itself, and also get a new access token to maintain authenticated connectivity to Databricks.

## OAuth Machine-to-Machine (M2M)

It is also known as the 2-legged OAuth and OAuth Client Credentials Flow (defined in OAuth 2.0 RFC 6749, section 4.4). The "two legs" refer to the direct interaction between the partner application (or DBSQL driver) and Databricks OAuth endpoint.

- **Use Case**: M2M interactions involve automated systems, services, or applications communicating with the Databricks platform without direct human intervention. This typically includes tasks like automating data ingestion, triggering data processing pipelines or synchronizing data between systems. M2M interactions are commonly used in scenarios where multiple systems or applications need to work together to achieve a desired outcome, such as ETL pipelines, data monitoring, and orchestration of complex workflows.

- **Interaction flow**


Partner Application
Databricks Auth Endpoint
DBSQL Driver
Databricks SQL Endpoint
Set OAuth Client ID/Secret with Service Principal ID/secret
SQL query
alt
[if access token expires or not exist:]
Request access token with Client ID/secret
Access token
SQL query + access token

A typical user-facing OAuth experience in a SaaS application using the M2M mechanism is:

1. A user logs in to the partner's website.

2. The user selects Databricks as the data source. The partner connects to Databricks using the pre-defined service principal and client secrets. The user does not perform explicit login or authentication operations for Databricks.

## Design[ UI ] Connector for Databricks in ISV Product/Application

When designing a connector to connect to Databricks using OAuth ersonal Access Tokens (PAT), you'll need to include several key fields. Whether you support OAuth U2M or/and M2M depends on your use case.

**Here are the essential fields for each authentication method:**

### Personal Access Token (PAT) Authentication

For PAT authentication, you'll need the following fields:

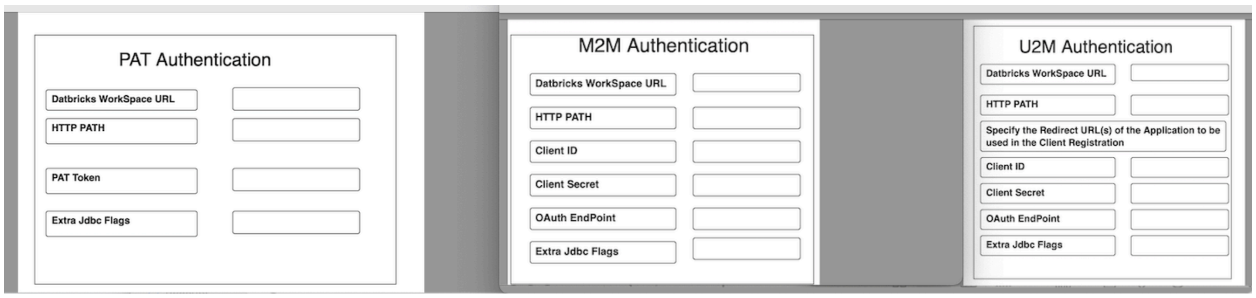
- **Hostname or Workspace URL**: The hostname for the workspace or the URL of your Databricks workspace (e.g., https://dbc-a1b2c3-d4e5.cloud.databricks.com).
- **Personal Access Token**: The token generated from the Databricks workspace.
- **HTTP Path**: The HTTP path to the Databricks SQL warehouse or Compute (optional, depending on your use case).

### OAuth Authentication

For OAuth authentication, you'll need these fields:

- **Hostname or Workspace URL**: The hostname for the workspace or the URL of your Databricks workspace (e.g., https://dbc-a1b2c3-d4e5.cloud.databricks.com).
- **HTTP Path**: The HTTP path to the Databricks SQL warehouse or Compute (optional, depending on your use case).
- **Connection type:** Per-User (U2M) or Service-Principal (M2M). This selection by the end user allows the connector to correctly choose the U2M or M2M OAuth flow (under the hood)
- **Redirect URL**:
    - **U2M** Partner needs to mention the URL to be used in the Client Registration
    - **M2M**: You don't need a Redirect URL
- **Client ID**: The ID of the OAuth application registered in Databricks.
- **Client Secret**: The secret generated for the OAuth application/Service-principle.
- Fields that can be set as **Advanced configuration**
    - Consider adding fields for additional JDBC parameters or custom key-value pairs
    - **Token Expiry**: **(Optional)** If there is a need for non-default token expiry.
- **OAuth EndPoint**: Either WorkSpace(optional) or Account Level.

- For **Workspace Level**: This can be auto-generated using the hostname or workspace URL. Please refer to the Appendix-I for WorkSpace-related endpoints
- For **Account Level:** Databricks Account URL needs to be provided. Please refer to Appendix-II for Account-related endpoints



## Getting Started

### Prerequisites

Before you begin, ensure you have the following:

- An active Databricks account and workspace
- The admin access to your Databricks account
- [OAuth U2M] IDP Integration is in place with the user either synced or created in Databricks

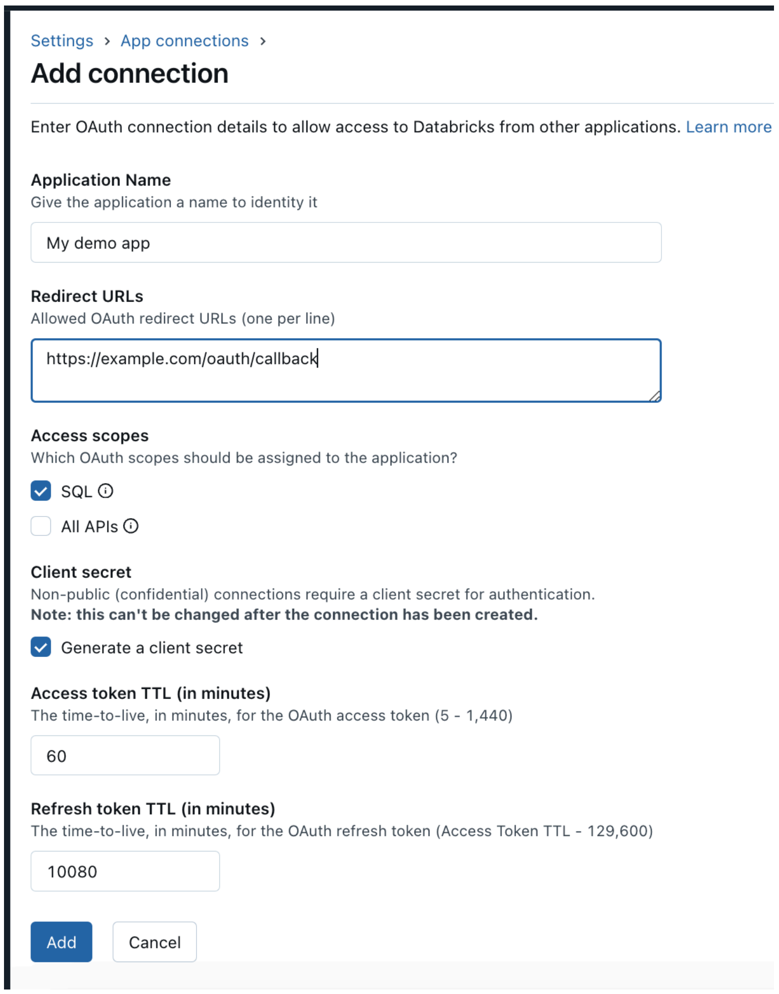
### OAuth U2M

#### Register OAuth Application

You can contact Databricks to create a default Databricks OAuth application or you can register an OAuth application for U2M in your account by following the steps:

1. Login to Databricks Account Console
   - AWS: https://accounts.cloud.databricks.com
   - Azure: https://accounts.azuredatabricks.net
   - GCP: https://accounts.gcp.databricks.com
2. Goto **Settings | App connections**
3. Click "**Add connection**"
4. Enter the application name
5. Enter the **Redirect URLs**, which should be the OAuth callback endpoint of your cloud application, see Implement OAuth callback endpoint in the application
6. Select "**All APIs**" in the "**Access scopes**" if your application needs to access other Databricks non-SQL APIs, otherwise leave as default
7. Click "**Add**" to create your OAuth application

8. A dialog "**Connection created**" will popup, please copy the "**Client ID"** and "**Client Secret**" in the dialog and store them somewhere as you won't be able to see the "**Client Secret**" again



Settings › App connections ›

Add connection

Enter OAuth connection details to allow access to Databricks from other applications. Learn more

Application Name
Give the application a name to identity it

My demo app

Redirect URLs
Allowed OAuth redirect URLs (one per line)

https://example.com/oauth/callback

Access scopes
Which OAuth scopes should be assigned to the application?

[✓ SQL]

All APIs

Client secret
Non-public (confidential) connections require a client secret for authentication.
**Note: this can't be changed after the connection has been created.**

[✓ Generate a client secret]

Access token TTL (in minutes)
The time-to-live, in minutes, for the OAuth access token (5 - 1,440)

60

Refresh token TTL (in minutes)
The time-to-live, in minutes, for the OAuth refresh token (Access Token TTL - 129,600)

10080

**Notes**:

- The following scopes are automatically granted to the application.
  - ***openid, email, profile:*** Required to generate the ID token.
  - ***offline_access***: Required to generate refresh tokens.
- It only registers the application in your account. You have to ask customers to register your application in their Databrick accounts if they want to use your application.

## Implement OAuth Authorization Code flow

A cloud-based partner application will need to implement the OAuth U2M flow (see above) to acquire an access token, that can then be used with the DBSQL driver.

To enhance security, we use PKCE (Proof Key for Code Exchange, defined in RFC 7636) in OAuth U2M flow. PKCE provides an additional layer of security by requiring the use of a dynamically generated code challenge and code verifier.

Here are the steps of the implementation.

**Step 1: Initiate OAuth authorization code flow**

When the application needs to access Databricks using the DBSQL driver, it needs to initiate the OAuth authorization code flow to acquire the access token.

**Step 1.1 Generate state, code_verifier, and code_challenge**

The application needs to generate the following values that will be used in the authorization code flow:

- **state**: A randomly generated string to maintain the state between the request and callback, protecting against CSRF attacks
- **code_verifier**: A high-entropy cryptographic random string used to generate the code challenge
- **code_challenge**: A Base64-URL-encoded SHA256 hash of the code_verifier

Here's an example of how to generate these values using Java:

```java
import java.security.SecureRandom;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public static String generateState() {
    SecureRandom secureRandom = new SecureRandom();
    byte[] stateBytes = new byte[32];
    secureRandom.nextBytes(stateBytes);
    return Base64.getUrlEncoder().withoutPadding().encodeToString(stateBytes);
}
public static String generateCodeVerifier() {
    SecureRandom secureRandom = new SecureRandom();
    byte[] codeVerifierBytes = new byte[32];
    secureRandom.nextBytes(codeVerifierBytes);
    return
Base64.getUrlEncoder().withoutPadding().encodeToString(codeVerifierBytes);
}

public static String generateCodeChallenge(String codeVerifier) throws
NoSuchAlgorithmException {
    MessageDigest digest = MessageDigest.getInstance("SHA-256");
    byte[] hash = digest.digest(codeVerifier.getBytes());
    return Base64.getUrlEncoder().withoutPadding().encodeToString(hash);
}
```

### Step 1.2 Direct the user to the authorization endpoint

To initiate the flow, the application should direct the user (browser) to the Databricks OAuth authorization endpoint: *https://<databricks workspace host>/odic/v1/authorize* with the following query parameters:

- **response_type**: ***code***
- **client_id:** client ID of the application created in Register OAuth Application
- **redirect_uri**: OAuth callback endpoint of the partner application, see Implement OAuth callback endpoint in the application
- **scope**:
    - ***sql offline_access***: Use this by default
    - ***all-apis offline_access***: Use this if your application needs to access Databricks non-SQL endpoint
    - ***sql offline_access openid email profile***: Use this if your application also needs to get an OAuth ID token that contains Databricks username, email, first and last name, and Databricks User ID. For example, some applications may want to validate the ID token to confirm the user's identity or access the user's profile (first and last name) in the token.
- **state**: state generated in step 1.1
- **code_challenge**: code challenge generated in step 1.1
- **code_challenge_method**: ***S256***

The user will be redirected to this endpoint to login and grant permission. After the user's authorization, the user will be redirected to **redirect_uri** with the authentication code and state parameters: *<redirect_uri>?code=<authorization code>&state=<state>*

### Step 2: Implement the OAuth callback endpoint in your application

To handle the authorization code redirect request from Databricks OAuth server, you need to implement an OAuth callback endpoint in your application and ensure this endpoint is accessible from the user and matches the **Redirect URL** of the OAuth application registered in Databricks.

Here is an example of an OAuth callback endpoint: *https://yourapp.com/oauth/callback*

Here is the recommended implementation of the callback endpoint:

1. Validate the **state** parameter in the request matches the **state** generated in step 1.1,
2. Send a **POST** request to Databricks OAuth token endpoint as below

```
Unset
POST https://<databricks workspace host>/odic/v1/token
 Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code
&code=AUTHORIZATION_CODE
&redirect_uri=YOUR_REDIRECT_URI
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
&code_verifier=YOUR_CODE_VERIFIER
```

**Parameters**:

- **grant_type**: ***authorization_code***
- **code:** the **code** parameter in the redirect (callback) request
- **redirect_uri:** URI of this OAuth callback endpoint
- **client_id**: client ID of the application created in Register OAuth Application
- **client_secret**: client secret of the application created in Register OAuth Application
- **code_verfier**: the code verifier generated in step 1.1

Response to the token request is like the following:

```JavaScript
{
  "access_token": "<OAuth access token>",
  "refresh_token": "<OAuth refresh token>",
  "token_type":"Bearer",
  "scope":"sql",
  "expires_in":3600
}
```

If the scopes "***openid email profile***" are included in the authorization request, the response will also include the ID token in the ***"id_token"*** field.

3. Get the access token, the refresh token; store the tokens along with the user session

## Create a DBSQL connection with a DBSQL driver and access token

Pass the access token obtained in the above step to the DBSQL driver to create a DBSQL connection as follows:

- JDBC Driver (**2.6.22** version or above)

```
Unset
jdbc:
databricks
://example.cloud.databricks.com:443/yourDatabricksHttpPath;AuthMech=11;Auth_Flo
w=0;Auth_AccessToken=YOUR_OAUTH_ACCESS_TOKEN
```

- ODBC Driver (**2.7.5** version or above)

```
Unset
Host=<server-hostname>;Port=443;HTTPPath=<http-path>;AuthMech=11;Auth_Flow=0;
Auth_AccessToken=YOUR_OAUTH_ACCESS_TOKEN
```

## Refreshing token

OAuth Access tokens are valid for a limited time (by default, 1 hour). For running new queries or for handling long-running queries, the cloud-based partner application must refresh the token in their business logic and set the new refreshed access token in the DBSQL driver.

Here is the recommended implementation of refreshing tokens.

1. Get the refresh token from the above token response
2. Send a **POST** request on Databricks OAuth token endpoint as below:

```
Unset
POST https://<databricks workspace host>/odic/v1/token
 Content-Type: application/x-www-form-urlencoded

 grant_type=refresh_token
 &refresh_token=YOUR_REFRESH_TOKEN
 &client_id=YOUR_CLIENT_ID
```

3. Get the access token from the response
4. Update the access token in the existing DBSQL driver connection as follows:

- JDBC driver

```
Java
Connection.setClientInfo("Auth_AccessToken", "YOUR_NEW_ACCESS_TOKEN")
```

Please note that as JDBC driver APIs are blocking the application may need to invoke the connection#setClientInfo() API on a different thread. If the token is expected to be valid for the time t, you can use a different thread which at time t/2 sets the refreshed OAuth access token to the JDBC driver.

- ODBC driver

```
C/C++
// Update the access token
char *credentials = "Auth_AccessToken=$(new token)"
SQLSetConnectAttr(dbc, 122, credentials, SQL_NTS); // 122 is Custom ODBC
property: SQL_ATTR_CREDENTIALS

// Refresh current connection
__int32 refreshMode = -1; // Refresh now
SQLSetConnectAttr(dbc, 123, reinterpret_cast<SQLPOINTER>(refreshMode),
SQL_IS_SMALLINT); // 123 is custom ODBC property: SQL_ATTR_REFRESH_CONNECTION
```

## OAuth M2M

### Create service principal and service principal secret

You need to create a Databrick service principal for your application M2M authentication. Here are the steps.

1. Login to Databricks Account Console
    - AWS: https://accounts.cloud.databricks.com
    - Azure: https://accounts.azuredatabricks.net
    - GCP: https://accounts.gcp.databricks.com
2. Select **User Management** from the left navigation panel.
3. From the **Service Principals** tab, click on **Add service principal**
4. (Azure only) Select "**Databricks managed**" under the "**Management**" section
5. Enter a name for the service principal and click on **Add**.
6. Select the service principal you just created.
7. Click on **Generate Secret**.
8. Copy the **Client ID** and **Secret** from the pop-up window and store them somewhere as you won't be able to see the "**Secret**" again
9. Assign the service principal to the workspace, see the instruction
10. Grant the service principal with the corresponding unity catalog privileges of the catalogs/schemas/tables which will be accessed by the application

### Create a DBSQL connection with a DBSQL driver

All the Databricks SQL drivers support OAuth M2M, you can create a DBSQL connection by passing the service principal client ID and the secret to DBSQL drivers.

Example connection URLs

```java
AuthMech=11;Auth_Flow=1;OAuth2ClientId=<custom_oauth_client_id>;OAuth2Secret=<s
ecret>;Auth_Scope=<default: sql>
```

## OAuth integration in the customers' accounts/workspaces

The instruction in the Getting Started section only enables the OAuth integration in the partner account, to make the partner application work with OAuth in every Databricks customer account/workspace, the partner application needs to

- For OAuth U2M: Support customized/configurable OAuth redirect URI, Client ID per tenant/customer
- For OAuth M2M: Support customized/configurable Service Principal Client ID/Secret per tenant/customer

### OAuth U2M

To enable the partner application OAuth U2M integration in the customer's Databricks account, the customer admin needs to:

1. Register OAuth application for the partner application, which is similar to the Registration Application in the partner's Databricks account
2. Store the client ID/Secret and redirect URI of the registered OAuth application in the partner application via its UI or API

To support the above scenario, the partner application needs to:

1. Expose UI/API that lets the customer store the client ID/secret and redirect the URI of the OAuth application registered in the Databricks
2. Store the mappings between the client ID/secret and customer tenant/account in the partner application
3. When initiating the OAuth authorization code flow or sending a token request, get the client ID/secret and redirect uri for the current session context by reading the above mappings and passing them to the authorization request

### OAuth M2M

To enable the partner application OAuth M2M integration in the customer's Databricks account, the customer admin needs to:

1. Create a service principal for the partner application, which is similar to [Create Service Principal in the partner's Databricks account]()
2. Grant the required permissions (defined by the partner application) to the service principal
3. Store the service principal client ID and secret in the partner application via its UI or API

To support the above scenario, the partner application needs to:

1. Expose UI/API that lets customer store service principal client ID/secret in the partner application
2. Store the mappings between the service principal client ID/secret and customer tenant/account in the partner application
3. Dynamically pass the service principal client ID/secret to the DBSQL driver

## Best Practices

### Persistence/caching of the tokens

The OAuth refresh token is long-lived. The user's OAuth refresh token should be persisted/cached in the business logic of the cloud-based partner application to ensure the user does not need to repeat the OAuth U2M re-login.

Cloud-based applications are typically used by multiple users simultaneously. Hence, the application should be able to persist OAuth refresh token per application user or session. For example, the user's OAuth tokens can be persisted on the cloud-based service side linked to their session.

One proposed persistence is to scope tokens such that for each (Databricks-workspace-host, user) tuple we store tokens independently.

databricks

## Appendix I: Databricks Workspace-level OAuth Endpoints

| Description | Endpoint URL |
|---|---|
| **OAuth Well-know configuration endpoint** | https://{databricks-host}/oidc/.well-known/openid-configuration |
| **OAuth Token Endpoint** | https://{databricks-host}/oidc/v1/token |
| **OAuth Authorization Endpoint** | https://{databricks-host}/oidc/v1/authorize |

## Appendix II: Databricks Account-level OAuth Endpoints

| Description | Endpoint URL |
|---|---|
| **Azure OAuth Well-know configuration endpoint** | https://accounts.azuredatabricks.net/oidc/accounts/<Databricks-AccountId>/.well-known/openid-configuration |
| **AWS OAuth Well-know configuration endpoint** | https://accounts.cloud.databricks.com/oidc/accounts/<Databricks-AccountId>/.well-known/openid-configuration |
| **GCP OAuth Well-know configuration endpoint** | https://accounts.gcp.databricks.com/oidc/accounts/<Databricks-AccountId>/.well-known/openid-configuration |
| **OAuth Authorization Endpoint** | https://<Cloud-Specifc-URL>/oidc/accounts/<Databricks-AccountId>/v1/authorize |
| **OAuth Token Endpoint** | https://<Cloud-Specifc-URL>/oidc/accounts/<Databricks-AccountId>/v1/token |

## Appendix III: OAuth Support in DBSQL Drivers

| Driver | Recommend versions for OAuth Token (U2M) | Recommend versions for OAuth M2M |
|---|---|---|

| | | |
|---|---|---|
| **JDBC** | **Latest (2.6.22 minimal)** | **Latest (2.6.22 minimal)** |
| **ODBC** | **Latest (2.7.5 minimal)** | **Latest (2.8.2 minimal)** |
| **Python Driver** | **Latest (2.1.1 minimal)** | **Latest (2.5.0 minimal)** |
| **GoLang Driver** | **Latest (1.5.2 minimal)** | **Latest (1.5.2 minimal)** |
| **NodeJS Driver** | **Latest (1.3.0 minimal)** | **Latest (1.5.0 minimal)** |

## Appendix IV: Links to sample code

Python Driver
https://github.com/databricks/databricks-sql-python/blob/main/examples/m2m_oauth.py