DRAFT - IT Project Guidance

OIDC Implementation

Version:

0.1

## Purpose

This document defines a consistent and accurate architectural pattern for using OpenID Connect (OIDC) to authenticate users and systems across a range of application types, including classic web apps, SPAs, mobile apps, and service-to-service integration.

## Synopsis

Many modern systems depend on federated identity to delegate authentication to external Identity Providers (IdPs). OpenID Connect, built on OAuth2, enables secure authentication while separating concerns between identity issuance and relying services. This guidance outlines when and how to establish clients, request tokens, validate identity, and manage user data responsibly. Appendices walk through each common pattern in detail.

## Contents

[Purpose 1](#_Toc201062122)

[Synopsis 1](#_Toc201062123)

[Contents 2](#_Toc201062124)

[Purpose and Audience 3](#_Toc201062125)

[Scope 3](#_Toc201062126)

[Background 3](#_Toc201062127)

[Overview of OAuth2 and OIDC 3](#_Toc201062128)

[Appendices 5](#_Toc201062129)

[Appendix A - Document Information 5](#_Toc201062130)

[Versions 5](#_Toc201062131)

[Images 5](#_Toc201062132)

[Tables 5](#_Toc201062133)

[References 5](#_Toc201062134)

[Review Distribution 5](#_Toc201062135)

[Audience 5](#_Toc201062136)

[Structure 5](#_Toc201062137)

[Diagrams 6](#_Toc201062138)

[Acronyms 6](#_Toc201062139)

[Terms 6](#_Toc201062140)

[Appendix A – Classic Web Multi-Page App (MPA) 7](#_Toc201062141)

[Configuration 7](#_Toc201062142)

[Use Case 7](#_Toc201062143)

[Appendix B – SPA with Backend APIs 13](#_Toc201062144)

[Appendix C – Mobile App 14](#_Toc201062145)

[Appendix D – Machine to Machine (M2M) 14](#_Toc201062146)

# Purpose and Audience

This document is written for developers, solution architects, and system integrators designing or implementing authentication flows. It assumes familiarity with HTTP, basic web architecture, and security concepts. It supports federated sign-in across enterprise or public identity providers.

# Scope

Covers identity authentication for:

* Traditional web interfaces using server-side rendering
* Browser-based single-page apps (SPA)
* Native mobile apps
* Machine-to-machine API clients

Does not cover account creation, identity proofing, or password policies of IdPs.

# Background

In federated systems, no one system owns user credentials. Each consuming system must instead:

* Register with the IdP (becoming a Client)
* Redirect unauthenticated users to the IdP
* Receive a token representing the authenticated user
* Use the token to retrieve user claims (attributes)
* Create or match local user records accordingly

This model reduces credential management risk but requires correct boundary enforcement and token handling.

# Overview of OAuth2 and OIDC

OAuth2 is a delegation framework that allows a client to obtain a token to act on behalf of a user or system.

OpenID Connect (OIDC) builds on OAuth2 by adding an identity layer, allowing the client to receive an ID Token containing identity claims about the authenticated user.

### Components

Key Components:

* **IdP (Identity Provider)**: Issues identity tokens.
* **Client**: Application requesting user authentication.
* **User Agent**: Browser or app where interaction occurs.
* **Tokens**: ID Token (identity), Access Token (API access), Refresh Token (optional).
* **Endpoints**: authorisation endpoint, token endpoint

## Standard OIDC Endpoints

OpenID Connect defines several key endpoints used in the authentication lifecycle. These are typically discoverable via the well-known configuration document provided by the IdP at:

/.well-known/openid-configuration

This returns a JSON object containing full URLs for the following endpoints:

* /authorize (Authorization Endpoint): for initiating user login and consent.
* /token (Token Endpoint): for exchanging authorization codes for tokens.
* /userinfo (UserInfo Endpoint): for retrieving user claims via access\_token.
* /logout (End Session Endpoint): for initiating logout and post-logout redirection.
* /.well-known/openid-configuration (Discovery Endpoint): for discovering all of the above.
* /jwks or similar (JWKS URI): for retrieving the IdP's public signing keys.

These endpoint paths may vary slightly per IdP but are provided via the discovery document.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).

Appendix A – Classic Web Multi-Page App (MPA)

### Prior Configuration

1. Register with IdP A *Svc A* as a client.
   * Get ClientId, ClientSecret.
   * Store these credentials securely in *Svc A*.  
     Preferably use secure storage, avoiding using on-server configuration text files, etc that can be read, or worse are added to the code base.

### Use Case

1. User A visits protected resource/page on *Svc A*.
   * Purpose: to view a protected resource (i.e., a secure webpage)
   * Steps:
     + App checks for session cookie.
     + No user record found.
2. Service redirects to IdP A:
   * Purpose: to initiate authentication and request user login and consent via the IdP.
   * Endpoint: /authorize (GET) – the Authorization Endpoint
   * Parameters (in URL Query):
     + client\_id = *Svc A*'s client id
     + redirect\_uri = URI to *Svc A* oidc capable handler endpoint (***not*** the protected resource) to which to return user after login, who in turn will redirect user to original protected resource after decoding that url from the scope that is roundtripped.
     + response\_type=code   
       This indicates that the client is using the *Authorization Code Flow*, which is the most secure OIDC pattern for server-side applications (such as MPAs). The value code signals that the IdP should return a temporary authorization code to the redirect\_uri, which the client will then exchange for tokens securely over a backend channel request. This avoids exposing tokens in the browser and enables confidential clients to authenticate safely.
     + scope=openid profile email (or minimal needed). The scope parameter defines which categories of user information the client is requesting access to.   
       openid is mandatory for all OIDC requests and indicates that this is an authentication request.   
         
       Work *hard* to ask for as little info as needed to get *Svc A* client (it is common misuse of OIDC to bloat the token with stuff never going to switch to/use). Stick with openid profile email.   
         
       Additional flags include:
       - profile: Requests basic profile information such as name, preferred username, and locale.
       - email: Requests access to the user's email address.
       - address: Requests physical address information.
       - phone: Requests phone number.
       - offline\_access: Requests a refresh token, allowing reauthentication without user interaction.  
         Note that if added, the IdP may prompt the user for additional consent, depending on its configuration.

Note that if added, the IdP may prompt the user for additional consent, depending on its configuration.

Be cautious when using vendor-specific or enterprise-defined scopes. These can introduce unexpected token bloat.

For example, Azure AD’s groups scope may inject full AD group membership lists, sometimes exceeding safe size limits for headers or cookies. Custom enterprise scopes—such as nsn (National Student Number) or edcontext (containing school, enrolment, or learner metadata)—may include structured JSON objects or arrays, often beyond what a relying party needs.

Overuse of such scopes leads to larger id\_tokens and access\_tokens, reduced performance, unnecessary data exposure, and increased risk of privacy leakage. Avoid treating the id\_token as a user profile dump. Instead, request only what is needed immediately. If additional attributes are required post-login, retrieve them later from the /userinfo endpoint using the access\_token, where claims can be policy-filtered, consent-governed, and returned in a more flexible manner.

* + - state = a base64 opaque string, often JSON that in turn contains at least a randomly generated value, cached in mem or session record for a short time (eg 15 minutes), and roundtripped here as an anti-CSRF token (think “correlationId”) but also the json can be used to round trip the returnUrl of the protected resource.
    - nonce = replay protection – this is a random value generated by the client when starting the authentication request. It is included in the ID token by the IdP and must match the original. Its purpose is to prevent replay attacks where an old or stolen ID token is reused fraudulently. The nonce must be stored temporarily (e.g. in session data or cache), scoped per user request, and expire after short use.
  + Response: html of login page (OK).

1. User Fills in IdP credentials form.
   * Purpose:
     + to enter validate User credentials
   * Steps:
     + Enters credentials   
       (e.g. IdP username (usually an email address) and password).
     + Submit for verification by IdP
2. IdP validates submitted credentials.
   * Purpose: to validate the submitted credentials
   * Outcome:
     + If validation fails, user is redirected to redirect\_uri with error query parameters such as error=access\_denied or error=login\_required to allow the client to handle failed login scenarios.
     + If validation succeeds, user is redirected to redirect\_uri?code=xyz&state=abc, where
       - state is the original CSRF token echoed back for verification (remember, it’s JSON that *also* contains the url to the protected resource page).
       - code is a one-time authorization code and
3. *Svc A* backend exchanges authorisation code for tokens:
   * Purpose: to submit retrieved code in order to get token in return.
   * Reason: The reason we received a code in the first place, rather then the token, was so that the token didn’t go through the web browser, which theoretically could have some javascript that was maliciously injected and could read and/or manipulate the token the server would accept. By passing the token only via back channel ensures fewer devices can intercept and/or change the token’s content.
   * Endpoint: /token (POST) - the Token Endpoint
   * Arguments:
     + ClientId and ClientSecret are sent using HTTP Basic Auth in the Authorisation Header:  
       Authorization: Basic bXktY2xpZW50OnMzY3IzdA==
   * Response:
     + The response is typically returned as a JSON payload in the response **body**, not as URL arguments.  
       {  
       "access\_token": "<opaque or JWT string>",  
        "id\_token": "<JWT string>",  
       "token\_type": "Bearer",  
       "expires\_in": 3600,  
       "scope": "openid profile email"  
       // and optionally:  
       “refresh\_token”: “<opaque string>”   
       }
     + id\_token: always a JWT containing identity claims. It is encoded using Base64URL and consists of three segments separated by dots: the header (metadata including signing algorithm), the payload (a JSON object containing the actual claims), and the signature (used to verify authenticity). This single string can be decoded to inspect its content but must be validated cryptographically. Base64URL encoding does not use dots—dots (.) are not part of the encoding. Instead, they act as separators between the three parts. Each part is a Base64URL-encoded JSON object (for header and payload) or a hash (for the signature). Used by *Svc A* only.

It contains iss (it comes regardless of scope sent as long as openid was used)

It contains sub (it comes regardless of scope sent as long as openid was used).  
It may also contain *email* and *username*, etc. depending on what scopes were sent and how the *IdP* was configured (as in hold info till you ask for it via user info endpoint in a second call).

* + - access\_token: opaque or JWT, used to later call the UserInfo endpoint or other APIs. This token is not the same as the first returned authorization code or the id\_token JWT.   
      It usually expires in 3600secs (one hour).
    - refresh\_token: opaque string. But only if ‘offline\_access’ was specified earlier, the IdP is configured to issue one, the client is confidential, the user accepted). The length is not defined by OIDC, however:
      * **Hours to days** for user sessions (commonly 24 hours, 7 days, or even longer).
      * **Until revoked**: some IdPs issue long-lived refresh tokens that last *indefinitely*, unless explicitly revoked (e.g. user logs out, password reset, admin action).
      * **Rotation policy**: some IdPs rotate the refresh token each time it's used, invalidating the previous one. This prevents replay if stolen. Plan always handling token changing or even being revoked.
  + Validation:
    - To confirm the token is authentic and has not been tampered with before trusting any claims inside it.

Most modern web frameworks (e.g. ASP.NET Core, Spring Security, Node's passport.js) handle token validation automatically, including fetching the JWKs, checking the signature, and validating basic claims. However, if not using such a library, then manually:

* Check the signature using the IdP's public key, which is typically published at a JWKs (JSON Web Key Set) URI. This is a standard endpoint (often at /.well-known/openid-configuration) where the IdP exposes one or more public keys in JWK format. The system fetching the JWT uses the kid (Key ID) in the JWT header to select the correct key from the JWKs and then uses that key to verify the signature's integrity and authenticity.
* then Verify claims :
  + iss (issuer – confirms the IdP identity),
  + aud (audience – must match the registered client ID that was given by *Svc A* when it was originally obtaining its credentials),
  + exp (expiration time – ensures the token is still valid), and nonce (replay protection – must match the original request to guard against replay attacks)

1. Parse claims from the returned id\_token:

* Purpose: to extract stable identifiers and useful display information for matching or creating a local user record.
* Notes:
  + claims will depend on scope given in first request.
  + However, even if profile or email is requested in the initial scope, the id\_token *may not include those claims*. This is normal and permitted by the specification. It will depend on how the IdP was configured.   
    For example, IdPs often minimise the id\_token for performance and security reasons, deferring claim release to the /userinfo endpoint. Additionally, some claims may be omitted if the user has not consented, if the IdP lacks the data, or if custom policies restrict access based on client type or context.   
    Always treat the id\_token as containing only essential identity information. If richer user attributes are needed, call /userinfo using the access token (see Step 8), although that will still be constrained by the scopes requested and granted in this step2 (getting access code).
* Steps:
  + sub: the unique, stable identifier assigned to the user by the IdP. This is persistent and non-reassignable, and is the correct value to use for identity matching. It is **not** an email address.
  + email: the user's current email address, if asked for in scope and consented to. It may change over time or be reused, and should not be used as a primary identifier.
  + name: a display name, often derived from the user’s full name as known to the IdP. This is for presentation and may not be stable.

1. Match or create local user.

* Purpose: To associate the authenticated identity with a persistent user record in the local system.
* Note
  + a User can have multiple Digital Identities associate to it.
* Operations:
  + Look up or insert a record in the Users table
  + Store a DigitalIdentity referencing the User record, linking the IdP (iss) and user identifier (sub) to your internal user id.
  + Example: DigitalIdentity { Idp = https://idpA.govt.nz, Subject = 'abc123', UserId = 42 }

1. Get more attributes:
   * Purpose:
     + The IdP may not include all user claims in the id\_token to reduce token size or to support dynamic consent (where the IdP only releases certain claims if the user explicitly consents to share them, either during initial login or via subsequent consent prompts).
   * Note: this is an optional step – but it’s almost always done because the initial token response may not include all claims permitted by the scopes requested in Step 2 (asking for the code).
   * Endpoint: /userinfo (GET or POST) - the UserInfo Endpoint
   * Arguments:
     + access\_token (as a Bearer token in the Authorization header)
     + No other parameters are typically required or permitted by standard-compliant IdPs. The access token alone authorises the request, and the IdP responds with a JSON object of claims associated with the user. Some IdPs may allow optional headers like Accept: application/json, but these are not necessary in conformant OIDC flows.
   * Returned:
     + The returned attributes are scoped by the original scope parameter in the authentication request. For example, if profile and email were requested, the /userinfo response may include name, email, preferred\_username, and similar fields.   
       The exact contents depend on what the IdP supports and what the user has consented to release. Always treat these attributes as subject to policy and consent constraints. The access token alone authorises the request, and the IdP responds with a JSON object of claims associated with the user. Some IdPs may allow optional headers like Accept: application/json, but these are not necessary in conformant OIDC flows.
   * Validate:
   * Match the returned nonce in the ID token against the one originally sent.
   * Match the returned state against the stored CSRF token value.
2. Establish session cookie.
   * Purpose:
     + To persist the authenticated user's login state across requests and pages after redirection from the IdP.
   * Method:
     + Store a cookie that will contain an identifier pointing to a session record in Sys A’s cache or database.
     + Always set the following flags on session cookies:

* Secure: ensures the cookie is only sent over HTTPS
* HttpOnly: prevents JavaScript from accessing the cookie
* SameSite=Strict or Lax: mitigates CSRF by limiting cross-origin sending
  + Considerations:
    - Before setting this up, the system designer must decide how long session validity should last. Session cookies may expire when the browser closes (session-based) or be configured with a specific timeout (e.g. 8 hours). Short durations improve security but may frustrate users. Longer durations improve usability but require caution—rolling expiration and periodic re-authentication help balance this.  
      If refresh is expected (e.g. to check current IdP state or retrieve updated claims), the system must retain the necessary information to reinitiate login. This includes the IdP’s authorization endpoint, the client ID, the scope, and optional parameters like prompt or max\_age. These values should be stored securely on the server (e.g. in user session metadata). They must not be stored in the user agent, as they can be exploited. Access and refresh tokens, in particular, must not be exposed in browser storage unless explicitly and safely managed.  
      Session cookies typically expire after a fixed period (20–60 minutes of inactivity or 8 hours total), as determined by system policy.

1. **(Optional) Refreshing the Access Token**This will acknowledge that in a classic MPA, the refresh token is often stored server-side and used only when needed. Here's a suggested insertion:  
     
   If the offline\_access scope was granted, and the IdP issued a refresh\_token, *Svc A* may use it to obtain a new access token without prompting the user to reauthenticate. This is useful in long-lived sessions where the access\_token has expired, but the local session cookie is still valid.  
     
   To use it, *Svc A* sends a POST request to the /token endpoint with:

grant\_type=refresh\_token

refresh\_token=<refresh\_token>

This is sent along with the client credentials. The response will contain a new access\_token, and optionally a new refresh\_token. Some IdPs rotate the refresh token on each use. Always plan to update or discard expired refresh tokens and reinitiate login if the refresh attempt fails.

Appendix B – SPA with Backend APIs

The Single Page Application (SPA) is a browser-based app that runs entirely on the client side. Because it has no secure server component of its own, it is classified as a **public client**. This means it **cannot securely store a client secret**, and therefore must use a protection mechanism called PKCE (Proof Key for Code Exchange) instead of supplying a shared secret when requesting tokens.

1. When the user attempts to access a protected feature in the SPA, and no existing valid session or token is found, the SPA initiates authentication.

It constructs a request to the IdP’s /authorize endpoint.

This request includes:

* client\_id: identifying the SPA as a registered application
* redirect\_uri: a location within the SPA to return the user to after authentication
* response\_type=code: signalling that the SPA wants an authorization code
* scope=openid profile email: requesting identity and basic claims
* code\_challenge and code\_challenge\_method: part of the PKCE protocol
* state and nonce: for CSRF protection and replay prevention

The user is redirected to the *Identity Provider (IdP)*, which displays a login page.

1. The user enters their credentials at the IdP. If authentication is successful, the IdP redirects the user’s browser back to the SPA’s redirect\_uri, including two URL parameters:

* code: the one-time authorization code
* state: which must be checked to match the value originally sent, ensuring the request was not intercepted or tampered with

1. The SPA now needs to exchange the code for tokens.

Because it cannot safely do this itself (as a public client), it sends the *code* and the original *code\_verifier* (from the PKCE setup) to a backend system—***Svc A***—that acts on its behalf.

*Svc A* is a confidential server-side client and **can safely hold client credentials** if needed.

This exchange must be done over a secure HTTPS connection.

1. *Svc A* makes a back-channel request to the IdP’s /token endpoint, submitting:

* The code received from the SPA
* The code\_verifier matching the earlier challenge
* The redirect\_uri (**must** match exactly)
* The client\_id (and optionally client\_secret, depending on IdP policy)

1. If the request is valid, the IdP responds with a set of tokens in a JSON object:

* id\_token: a JWT containing the user’s identity claims (e.g. sub, name, email)  
  Note that iss and sub always come (as long as you used openid as a scope).   
  name and email depend on scopes asked, and how the IdP is configured (as in hold it till requested later)
* access\_token: used to access other APIs
* refresh\_token: **only if the offline\_access scope was requested and permitted**. This token allows the backend to request new access tokens later, without the user logging in again.

1. *Svc A* validates the tokens it received.

It checks:

* The cryptographic signature of the id\_token
* The *issuer* (iss),
* *audience* (aud),
* *expiration* (exp), and
* replay protection (nonce)

Once validated, it creates or updates a local user record as needed, and establishes a server-side session.

It then sends a secure response to the SPA, such as a short-lived session identifier or user display info, *without exposing any tokens* to the browser.

1. Now authenticated, the SPA begins making API calls to *Svc A* on behalf of the user.

For each call, it includes the access\_token in the HTTP Authorization header as a Bearer token:   
Authorization: Bearer <access\_token>

1. *Svc A* validates the token on each request, ensuring it is unexpired, issued by the expected *IdP*, and intended for this *client*.
2. **Access Token Expiration and Refresh**

Access tokens typically expire after a short time—usually one hour. Once expired, the SPA cannot use the old token to make further API calls.

If a refresh\_token was issued and securely stored by *Svc A* during Step 5, it can now be used to request a new access token without requiring the user to go through the login process again. This is called a **refresh flow**.

The steps are as follows:

1. The *SPA* makes an API call to *Svc A*, but the access\_token has expired. *Svc A* responds with an *HTTP 401 (Unauthorized)* or similar failure.
2. Rather than prompting the user to log in again, *Svc A* checks if it has a valid refresh\_token stored from the earlier login.

If so, it prepares a new request to the /token endpoint of the IdP.

1. *Svc A* sends a POST request with the following form data:

grant\_type=refresh\_token

refresh\_token=<stored\_refresh\_token>

client\_id=<client\_id>

client\_secret=<client\_secret> (if required)

1. If the refresh token is still valid, the IdP will return a new access\_token, and sometimes a new refresh\_token as well (depending on the IdP’s rotation policy). *Svc A* updates its stored token accordingly.
2. The original API call can now be retried using the new token, completing successfully without user disruption.  
     
   If the refresh token has expired, been revoked, or is otherwise rejected, *Svc A* must fall back to requiring the user to reauthenticate via the IdP.

This refresh pattern allows long-running *SPA*s to remain functional over time without repeatedly interrupting the user for login—while still ensuring that tokens are short-lived and revocable, as a security best practice.

Appendix C – Mobile App

1. Native app redirects via in-app browser (or secure web view) to IdP Authorization Endpoint.
2. Uses PKCE and redirect\_uri to custom scheme (e.g. myapp://callback).
3. IdP redirects to app’s URI with code.
4. App exchanges code for tokens via secure backend or direct call.
5. Receives id\_token, validates locally.
6. Calls backend APIs with access\_token.
7. Token refresh managed with Refresh Token grant or silent login.

Appendix D – Machine to Machine - Incoming (M2M)

This pattern applies when *Svc A* provides an API that other systems or services call in a machine-to-machine (non-user) context.

These other systems authenticate themselves using access tokens issued by a mutually trusted Identity Provider (IdP).

There is no human user or browser involved.

*Svc A* must act as a resource server, meaning it protects its endpoints by validating the incoming access tokens.

1. ***Svc A* is Registered as a Resource Server with the IdP**

To accept and validate access tokens, *Svc A* must be aware of the Identity Provider’s configuration. This includes:

* The issuer URI (e.g. https://idp.govt.nz)
* The set of **public keys** used by the IdP to sign access tokens (usually provided via a **JWKS endpoint**)
* The expected structure and content of tokens, including the aud (audience), scope, and other claims

*Svc A* does not require a client\_id or client\_secret in this role—it is not initiating token requests, only validating tokens presented to it.

1. **A Third-Party System Calls *Svc A*’s API with an Access Token**

The calling system (e.g. Svc B) must first authenticate with the shared IdP using the **Client Credentials Grant**, as described in Appendix D. If successful, it receives an access\_token, which it includes in requests to *Svc A*:  
Authorization: Bearer <access\_token>

*Svc A* receives this request and extracts the token from the Authorization header.

1. ***Svc A* Validates the Incoming Access Token**

*Svc A* must treat all access tokens as *untrusted until validated*.

Depending on the *IdP* and token *type*, validation can take two forms:

* For JWT tokens (which are self-contained):
  + Decode the token
  + Verify its *signature* using the IdP’s published *JWKs*
  + Check standard claims:
    - iss (*issuer*): must match the known IdP
    - aud (*audience*): must include *Svc A*’s identifier or API name
    - exp (*expiry*): must not be expired
    - Any required scope or custom claims (e.g. roles or purpose-of-use)
* For opaque tokens (which cannot be decoded):
  + Call the *IdP*’s /introspect endpoint, passing the token
  + Endpoint: /introspect (TODO)
  + Response: a response indicating whether the token is *valid*, *expired*, or *revoked*,
  + If *valid*, extract *scopes* or *claims*

Note: *Svc A* must fail closed. If any validation fails (invalid signature, missing scope, unknown issuer, expired token), the request is rejected with a 401 Unauthorized.

1. ***Svc A* Applies Authorisation Rules Based on Token Claims**

After validating the token’s authenticity, *Svc A* must apply *authorisation* logic to determine whether the caller is permitted to access the requested resource.

This may be based on:

* scope claims (e.g. read:students, write:attendance)
* client\_id (identifying which system made the request)
* sub (subject), if present
* *Custom claims* (e.g. organisation ID, integration level)

The scopes granted by the IdP to the calling system must match the required scope for the endpoint being accessed. If not, return 403 Forbidden.

1. ***Svc A* Processes the Request and Returns a Response**

If the request is authorised, *Svc A* proceeds to process it like any internal request and returns the appropriate response.

*Svc A* should always log:

* The caller’s identity (client\_id or sub)
* The requested path and method
* Whether the token was valid and authorised
* Any errors encountered in validation

These logs are important for audit and monitoring.

1. **Token Expiry and Retry Logic**

Since M2M access tokens are short-lived (typically 5 to 60 minutes), *Svc A* will sometimes receive expired tokens. The standard practice is:

* Reject expired tokens with 401 Unauthorized
* Do not attempt to refresh the token—***Svc A* is not the client**
* The caller (e.g. Svc B) is responsible for requesting a new token and retrying

If too many requests arrive with expired or invalid tokens, this may indicate poor client-side token management or clock drift.

1. **Security Considerations**

* Never accept tokens from untrusted IdPs
* Never assume token validity based on format alone—**always validate**
* Use JWKS caching to minimise latency but respect key rotation
* If using opaque tokens, ensure the /introspect endpoint is secure and performant
* Apply least privilege: do not accept broad scopes for sensitive operations

This model is common in ecosystem or cross-organisation integrations, where multiple systems rely on a shared IdP or federation trust model. *Svc A*, as the accepting API, must act as a **robust gatekeeper**, ensuring that only valid, authorised requests are allowed through.

Appendix E – Machine to Machine - Outgoing (M2M)

This pattern describes a scenario where one system (Service A) needs to access another system (Service B or an API gateway) in a **non-interactive** context—meaning, there is **no user involved**. The access is performed using **client credentials** only, typically for background processing, scheduled jobs, integrations, or system-level automation.

1. ***Svc A* Registers as a Confidential Client with the Identity Provider**

To participate in machine-to-machine authentication, **Service A (*Svc A*)** must first be registered as a **confidential client** with the Identity Provider (IdP). As part of this registration, *Svc A* is issued:

* A client\_id: a public identifier for the client application
* A client\_secret: a sensitive shared secret used to authenticate the client to the IdP

These values must be stored securely.   
Unlike public clients (such as *SPA*s), confidential clients like *Svc A* *are* permitted to hold secrets because they operate in controlled, non-user-facing environments.

1. ***Svc A* Initiates Authentication Using the Client Credentials Grant**

When *Svc A* needs to call another system—usually an API protected by the IdP—it must first obtain an **access token** to prove its identity.   
It does this by initiating a *Client Credentials Grant* flow.

This flow does **not involve a user**, login screen, redirect, or browser.   
It is a direct server-to-server exchange.

1. ***Svc A* Calls the Token Endpoint**

*Svc A* prepares and sends a POST request to the IdP’s /token endpoint. The request includes:

* The grant\_type=client\_credentials, which signals that this is a machine-to-machine exchange, not a user login.
* The client\_id and client\_secret for authentication.
* An optional scope parameter, specifying what permissions *Svc A* is requesting (e.g., read:data, write:invoices).

The request may look like this (sent as application/x-www-form-urlencoded):

POST /token

Host: idp.govt.nz

Authorization: Basic <base64(client\_id:client\_secret)>

Content-Type: application/x-www-form-urlencoded

grant\_type=client\_credentials

scope=read:data write:data

Alternatively, client\_id and client\_secret may be passed in the body if the IdP does not use HTTP Basic auth.

1. **The IdP Responds with an Access Token**

If the request is valid and the client is authorised, the Identity Provider returns a response containing an access\_token, and possibly additional metadata. For example:

{

"access\_token": "<opaque or JWT string>",

"token\_type": "Bearer",

"expires\_in": 3600,

"scope": "read:data write:data"

}

* The access\_token is usually a short-lived bearer token, often a JWT, but could also be opaque.
* No id\_token or refresh\_token is returned in this flow, because no user is involved and no refresh is permitted under the base OAuth2 specification (unless extended by vendor-specific logic).

1. ***Svc A* Uses the Access Token to Call APIs**

*Svc A* includes the *access token* in its HTTP requests to the protected API or service.   
For example:

GET /api/v1/data

Authorization: Bearer <access\_token>

The receiving system validates the token using standard means:

* If *JWT*: by checking its signature, expiry, issuer, and scopes
* If *opaque*: by introspecting the token via the IdP

Access is granted or denied based on:

* The scopes requested and then granted by the IdP
* The roles or permissions associated with the client\_id
* The validation rules enforced by the resource server (e.g. Svc B)

1. **Token Expiration and Renewal**  
   *Access tokens* issued via the *Client Credentials Grant* are typically short-lived —often 5 to 60 minutes, depending on policy.   
     
   Once expired:

* *Svc A* must initiate a new token request by repeating Step 3.
* There is **no refresh token**. The token cannot be refreshed—only reissued.
* There is no user session to track, so the client service must manage its own retry, caching, and error-handling logic.

Notes:  
It is common for M2M systems to:

* Cache the token in memory while it remains valid
* Proactively renew the token a few minutes before expiry
* Retry failed API calls once if a 401 (Unauthorized) is returned, fetching a new token before attempting again

This pattern ensures security by keeping tokens ephemeral, limits exposure in case of compromise, and eliminates the need for long-lived credentials on either side of the transaction.

Reasoning Notes:

* Tokens are kept light to reduce exposure and parsing overhead.
* ID token is for client use only, not for API.
* Access token is bearer and must be secured in transit.
* Claims not always embedded to support consent minimisation and attribute revocation.
* Local user record enables role assignment, display formatting, and session continuity.