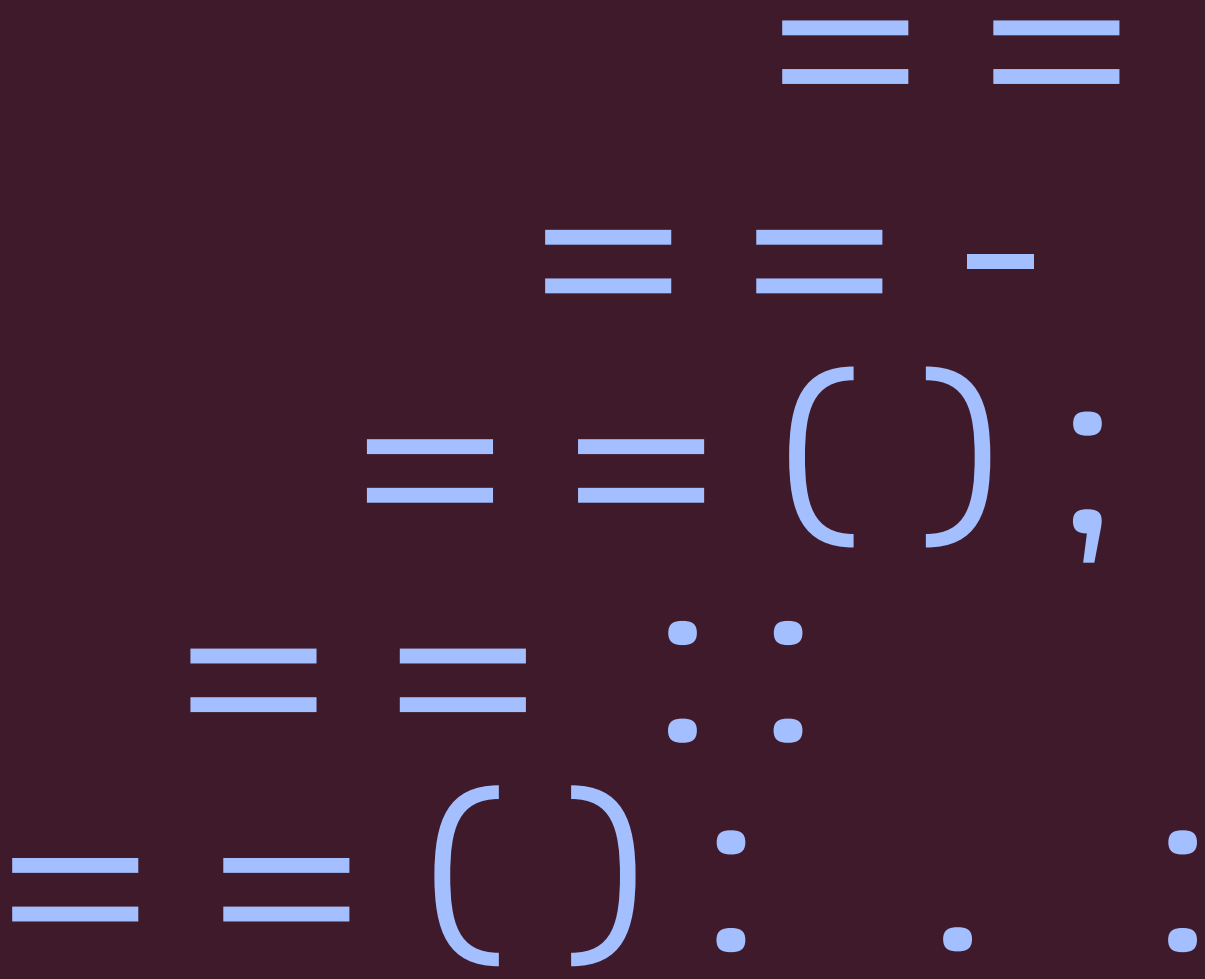
Agency for Digital Government

OIO OpenID Connect Profiles

Version 1.0
Draft 2



18.05.2026

Contents

1. Introduction

1.1 Preface

The Danish Agency for Digital Government has historically developed interoperability profiles for federated identity management to be used across the public sector. The goal is to promote interoperability, ensure a consistent and high level of security, and enable the reuse of implementations.

A key element is the profiling of modern authentication and authorization services based on OpenID Connect, JWT, and OAuth, which can be reused as common building blocks across many business applications, clients, and APIs. These enable authentication of end users based on [NSIS] or [eIDAS] levels of assurance, as well as subsequent authorization of the client (e.g. an app or web client) based on end-user consent, followed by the issuance and management of security tokens—in many ways similar to existing services based on OIO SAML and WS-Trust.

A foundational building block is the establishment of the necessary specifications and profiles that ensure interoperability and a high level of security. This document contains deployment profiles for OpenID Connect [OIDC] and OAuth 2.0, detailing the protocols for interaction between a client, an Authorization Server, a Token Server, and external APIs that consume tokens. The specifications are written with NemLog-in in mind but can also be used elsewhere.

The profiles can be used with various types of clients: native apps, web applications with a backend, JavaScript applications without a backend, as well as other client types with similar requirements.

1.2 Audience

The document is written for a technical audience including architects, security professionals and developers already familiar with OAuth 2.0, OpenID Connect, JWT, REST, TLS and other related technologies and specifications.

1.3 Usage Scenarios

This profile is intended for use within Danish public sector federations where information about authenticated identities is federated across service providers. The goal is to achieve standardization, interoperability, security and privacy, while enabling re-use of common implementations. Service providers in the public sector should be provided with common authentication and authorization services reducing the need for local implementations.

The current version focuses on the most common scenarios involving native apps and web apps. More advanced use cases may however be added later – including scenarios with federated Authorization Servers or APIs exchanging incoming tokens for downstream invocation of other APIs.

1.4 Overview of Specifications

A set of specifications and documents are developed, covering various aspects:

- This document covers protocols for the interaction between a client and an OIDC Authorization Server, a token exchange endpoint and token consuming APIs. The goal is to issue tokens to a client so they can establish a local session and access an external API.
- The OIO JWT Token Profile [OIO JWT] specifies formats for JWT tokens used with this profile, including claims, privileges and signatures. It is inspired by the OIO SAML 4 Web SSO profile [OIOSAML] and OIO Basic Privilege Profile [OIO-BPP].

The specifications are independent of NemLog-in and can be used everywhere to promote interoperability. In particular, early local implementations can use them in order to pave the way for a smooth transition from a local to a central implementation provided by NemLog-in. This approach thus minimizes the risk of redoing the client or API implementation at a later stage.

2. Notation and terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the following typographical conventions in text: `<ns:Element>`, `Attribute`, **Datatype**, `OtherCode`. The normative requirements of this specification are individually labeled with a unique identifier in the following form: **[OIO-EXAMPLE-01]**. All information within these requirements should be considered normative unless it is set in *italic* type. Italicized text is non-normative and is intended to provide additional information that may be helpful in implementing the normative requirements.

2.1 Terminology

This specification describes flows involving the following actors:

- **Client** – a native app, browser app or system client acting as a client in OAuth and OpenID Connect sense. It provides application services to the end-user, requests access tokens and consumes one or more external APIs e.g. for retrieving or updating data about the end-user. Clients are considered 'confidential' when the client application can securely store credentials (such as a client secret or private key) because it runs in a trusted environment.
- **SP API** – Service Provider API. An API offered by a Service Provider which is protected by a trusted Authorization- and Token Server – i.e. all API access requires presentation of a signed security token. The API Service Provider can be the same or a different organization providing the client. The API provider is referred to as a WSP (Web Service Provider) in the OIO IDWS infrastructure.
- **End-user** – a person authorizing client access on his/her behalf regarding defined scopes, and in case the client is a native app installs the app on their personal mobile device.
- **Authorization Server** – a central OAuth 2.0 infrastructure component.
- **Token Server** – OIDC/OAuth 2.0 infrastructure component issuing tokens which provide access to external APIs as well as the internal APIs (UserInfo endpoint).

2.2 Client types

Several types of clients can be used with this profile:

a) **Native apps** installed on the end-user device which may gain access to external APIs via tokens. These will be considered public clients as defined in OAuth 2.0. It is assumed that the device is personal and that access and refresh tokens for the end-user can be securely stored on the device; if this is not the case, one of the other variants below should be used.
b) **Enhanced native apps** are native apps that with additional measures can act a confidential client e.g. using a back-end (e.g. using the Backend for Frontend (BFF) pattern) or app

attestations[1] combined with dynamic client registration[2]. Enhanced native apps can authenticate using keys and tokens either protected in a back-end or secure local storage.

c) **Web Applications with a backend**. In these applications, JavaScript code is loaded from a dynamic Application Server that also has the ability to execute code itself. It is assumed that the Application Server performs the OAuth and OIDC interactions itself and keeps tokens stored internally, creating a separate session with the browser via a traditional session cookie - see [BBA] for additional details. The Application Server (backend) will be considered a confidential client for the purposes of its OAuth interactions.

d) **JavaScript Applications** without a backend (also known as 'Single Page Applications'). Here the entire application runs in the browser, and the client should therefore be considered a public client. Note that to be able to interact with Authorization Servers and Token Servers from different domains, these must support the necessary CORS headers in order to avoid same-origin restrictions imposed by browsers.

e) **System clients** are backends that invoke APIs provided by other backends and are considered confidential clients

Other types of clients can operate using the same mechanisms described above. Since client capabilities vary, the requirements stated will differ among them. Generally, clients should use the best available security mechanisms, so e.g. confidential clients should use client authentication and pushed authorizations requests (PAR). It is up to local deployments to determine which client types are allowed in a specific context since that will require trade-offs between security vs functionality. Some deployments may only allow confidential clients where others may need to support public clients as well.

The vast majority of the flows and requirements in this profile apply to all types of clients, and where requirements do depend on the client type, it will be stated explicitly. The illustrations and drawings primarily show examples of clients being native apps, since this scenario has been the main reason for writing this specification.

The table below shows important properties of the main client types which will be explained in subsequent chapters:

[1] E.g. DeviceCheck, SafetyNet, key attestations etc.

[2] Dynamic client registration is not profiled in this document as it will be highly implementation specific.

Client type	Native app	Enhanced Native app	Web/JS app with a Backend	JS app without a Backend
Allowed flows	OAuth 2.0 Authorization code flow	OAuth 2.0 Authorization code flow	OAuth 2.0 Authorization code flow	OAuth 2.0 Authorization code flow
PKCE	Mandatory	Mandatory	Mandatory	Mandatory
Token storage	Secure device storage	Secure device or back-end	In backend or encrypted cookie in browser	Browser APIs
Authorization Server Requirements	Redirect URI registered and exact match required; no wildcards allowed	Redirect URI registered and exact match required; no wildcards allowed	Redirect URI registered and exact match required; no wildcards allowed	CORS headers enabled; Redirect URI registered and exact match required; no wildcards allowed
Refresh Token Policy	Long-lived refresh tokens allowed (not expiring) if revocation mechanism i and token rotation is implemented	Long-lived refresh tokens allowed (not expiring) if revocation mechanism and token rotation is implemented	Refresh tokens allowed up to 8 hrs; must be invalidated on logout	Refresh tokens allowed up to 60 min and ONLY with rotation on each use.
Client Authentication	Not possible (public client); use re-direct URI as mitigation	Mandatory	Mandatory	Not possible (public client); use re-direct URI as proof
Pushed Authorization Requests	Optional	Mandatory	Mandatory	Optional
Other security req			Limit JavaScript execution to set of defined origins	Limit JavaScript execution to set of defined origins
Logout handling	No session (only refresh token revocation).	No session (only refresh token revocation).	Backend should send/receive logout request and invalidate session cookie and tokens.	App should poll Authorization Server (if possible) via a frame to detect session termination and invalidate tokens.

2.3 Token taxonomy

This profile distinguishes between:

1. ID Tokens issued to the client for user authentication,
2. Access Tokens issued for the UserInfo endpoint,
3. Refresh Tokens used to obtain new tokens, and
4. Delegated Access Tokens issued through OAuth 2.0 Token Exchange for presentation to external SP APIs.

3. Scenario overview

The figure below illustrates the main components and their interactions a native-app client is authorized by the user and subsequently invokes an API protected by the Authorization Server.

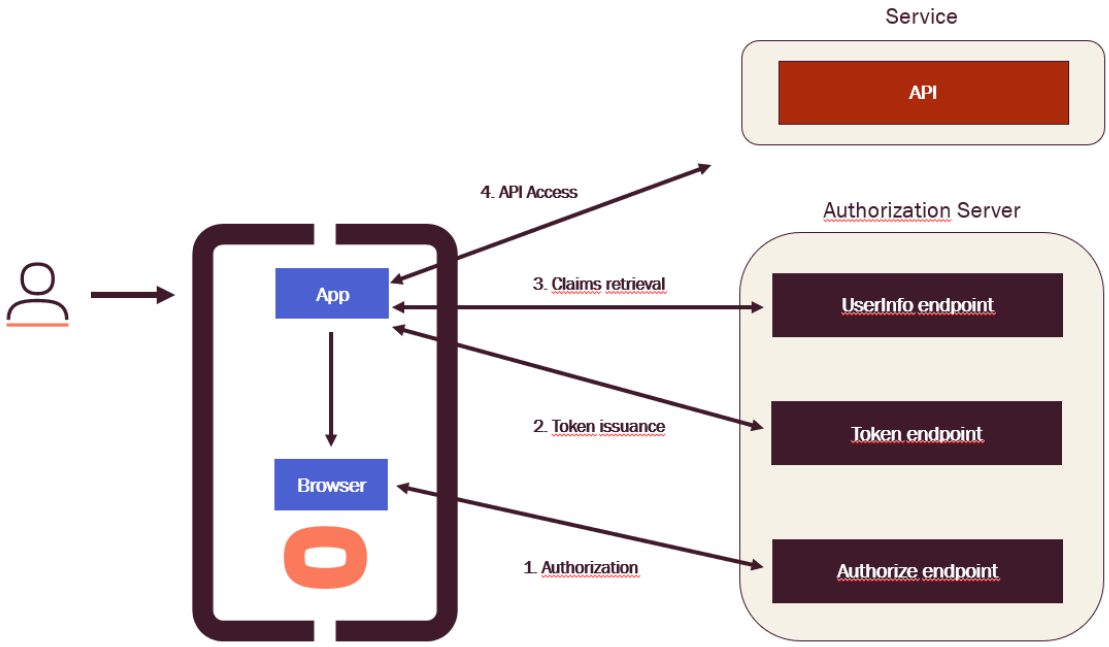


Figure 1: Main scenario with four phases

The main phases are:

1. The client initiates an authorization request by redirecting the user's browser to an Authorization Server via a separate user agent (e.g. browser). The client authenticates and uses pushed requests (PAR) if supported by the client type. The Authorization Server ensures authentication of the end-user (possibly by using a downstream Identity Provider). After authentication, the user is prompted to approve any scopes in the request that are marked as user-granted scopes. If authorization is successful, the Authorization server issues an authorization code (according to the OAuth authorization code flow)
2. The client exchanges the authorization code for a set of tokens: an ID Token to be consumed within the client App, an Access Token to be used with the UserInfo endpoint and a Refresh Token also used with the Token endpoint. The client may also request a token for an external API via a token exchange.
3. The client may request claims about the user from the UserInfo endpoint using the access token and retrieves a signed token with the requested claims (provided user approval during the authorization phase).
4. The client accesses an external API by presenting a valid security token issued by the Token endpoint. The security token provides the claims necessary for fulfilling the API's access control policy.

All tokens except Refresh Tokens are short-lived which ensures that they are renewed often and therefore get updated frequently. More details on this are described in chapter 5.

3.1 Pre-requisites

A number of pre-requisites are assumed to be in place before the above flows can be executed:

- The end-user has obtained relevant credentials needed for authentication (e.g. MitID or credential from local IdP).
- If the client is a native app, the end-user has a personal device[3]. Other flows should be used for non-personal devices where security tokens for a specific user cannot be persisted on the device. The native app has been installed on the end-user device (e.g. from a public app store or a closed app store).
- The client and SP API have been registered and configured with the central Authorization Server including relevant parameters / metadata:
    - Relevant identifiers (e.g. EntityIDs) have been assigned for client and API instances such that they can be referenced in tokens and protocol messages.
    - The type of client has been registered (see section 2.1 for details).
    - The client must have registered a unique **redirect URI** for returning the authorization response to the client, and the URI scheme should be based on a domain name that is under the control of the service provider of the client. More details are described in chapter 4.
    - Necessary privileges and scopes have been defined for the client and SP API such that they can be requested by the client and issued in tokens. See section 3.2 for details.
    - Metadata and certificates have been exchanged in advance as part of trust establishment.
        - Native app clients have been configured with trust anchors.
        - Clients have been configured with trusted token signing certificates in order for token signatures to be securely validated.
        - Confidential clients have registered their secret or certificate for client authentication.
    - Web clients must register a logout URI as part of the client registration process.

The actual registration process which establishes these pre-conditions will not be described in this document as it is highly implementation specific. For example, in NemLog-in the registration will be based on the existing administration portal.

3.2 Authorization models

This section describes the different authorization models. A central design goal has been to reuse the existing mechanisms for web applications and SOAP web services (in particular the OIO Basic

[3] The profiles assume that security tokens can be persisted in Apps including a long-lived Refresh-token. The risk profile of the App and user terms may define whether it is acceptable to do this on a device shared in a family or shared among colleagues in workplace environment.

Privilege Profile) such that API providers can reuse existing logic and access control policies. Other authorization models can be added later if needed.

3.2.1 Scopes granted by end-user

A fundamental design principle in OAuth (and hence this profile) is that end-users are able to authorize scopes granted to the client by authenticating to the Authorization Server and providing explicit consent for the client to act on their behalf. This is accomplished by including a set of OAuth scopes in the authorization request from the client, which allows the Authorization Server to obtain the necessary consent from the end-user and reflect it in issued tokens. This consent both covers the authorization to obtain user tokens with attributes as well as authorization to invoke external APIs (SP API). In this profile, scopes are both used to specify desired attributes and desired access rights.

Therefore, the authorization request (see next section) has to contain sufficient scopes to cover all APIs and scopes which the client needs to invoke on the end-user's behalf. If the client at a later stage needs further access (new API or scope), a fresh authorization request is required with the additional scopes added – which the user can then consent to.

As specified in the JWT Token Profile [OIO JWT], Delegated Access Tokens for SP APIs will contain privileges according to the model defined in OIO Basic Privilege Profile. Privileges are URIs defined by a Service Provider representing a specific access with that Service Provider. Thus, the meaning, granularity and consent text of privileges is defined entirely by the Service Provider - the infrastructure is just a mediator.

It is assumed that privileges to be requested and asserted in tokens will be registered in advance with the Authorization and Token Server as indicated in the example below:

Privilege info	Example values of privilege metadata registered
SP EntityID	https://ngdp.digst.dk
Privilege URI	https://ngdp.digst.dk/priv/read_mail
OAuth scope shorthand[4]	xq7j
Description	This privilege grants access to read mail from a citizen inbox in the Digital Post solution.
UI Context text (DK)	"Vil du give samtykke til, at denne App tilgår din Digitale Post fra det offentlige?"

The Authorization Server registration process ensures uniqueness of EntityIDs, privileges URIs, scope shorthands etc. and ensures proper ownership (e.g. an admin can only administer relevant Apps and APIs).

Thus, if the client includes the `xq7j` shorthand in the scope parameter as part of its request (see chapter 4), the Authorization Server will prompt the end-user for consent to authorize the client to access their mail in the Digital Post solution, and if consent is granted, the client will subsequently be able to obtain an Access Token for the relevant SP API, where the associated privilege URI `https://ngdp.digst.dk/priv/read_mail` is included with scope of the citizen.

Below is given an example of the resulting JSON structure within the Access Token based on the OIO JWT Profile [OIO JWT], where the privilege is included with scope[5] of "1202801024" (CPR number of citizen):

```
{
    "privilegegroups" : [
       {
           "privileges" : ["https://ngdp.digst.dk/priv/read_mail"],
           "scope" : "urn:dk:gov:saml:cprNumberIdentifier:1202801024"
       }
    ]
}
```

A similar model can be used for other scopes and for data restrictions; see the OIO JWT Profile [OIO JWT] for details.

[4] In order to keep requests small enough to fit in HTTP headers used with the OIDC authentication request, privileges are suggested to have a unique short-hand such that the entire URI is not necessary.

[5] Note that the scope in OIO Basic Privilege Profile should not be confused with scope in OAuth / OIDC. The first is the context of a privilege (e.g. person or organization the privilege applies to) and the latter corresponds to a given access requested (similar to a privilege in OIO BPP).

3.2.2 Scopes granted by API provider (WSP)

In addition to scopes granted by the end-user mentioned in the previous section, the Authorization Server may allow the API provider to grant privileges/scopes to certain clients independent of the user.

This is similar to the current mechanism in the NemLog-in STS, where a WSP (Web Service Provider) can define a number of privileges, which can then be granted to certain Web Service Consumers – i.e. clients of the WSP. The assignment of privileges is done by the WSP administrator in the NemLog-in administration portal.

A similar model can be used with clients and APIs – acting as WSC and WSP respectively. It can be used to grant specific access only to certain clients.

This model requires that the Authorization Server is able to securely authenticate the client.

Privileges granted by the API provider will be represented in the same way as user-granted privileges except they have a different scope being the client ID instead of the end-user identifier (e.g. CPR):

```
{
    "privilegegroups" : [
       {
           "privileges" : ["https://ngdp.digst.dk/priv/read_mail"],
           "scope" : "https://digst.dk/ngdp/apps/borger_dk_client"
       }
    ]
}
```

4. Client Authorization Profile

This chapter specifies a profile of OpenID Connect (using the [OAuth] 2.0 authorization code grant) used for the initial authorization of the client. The profile is primarily based on [OIDC], [OAuth] and [RFC8252].

The figure below illustrates the individual steps covering phase 1 – 3 in the main scenario from Figure 1.

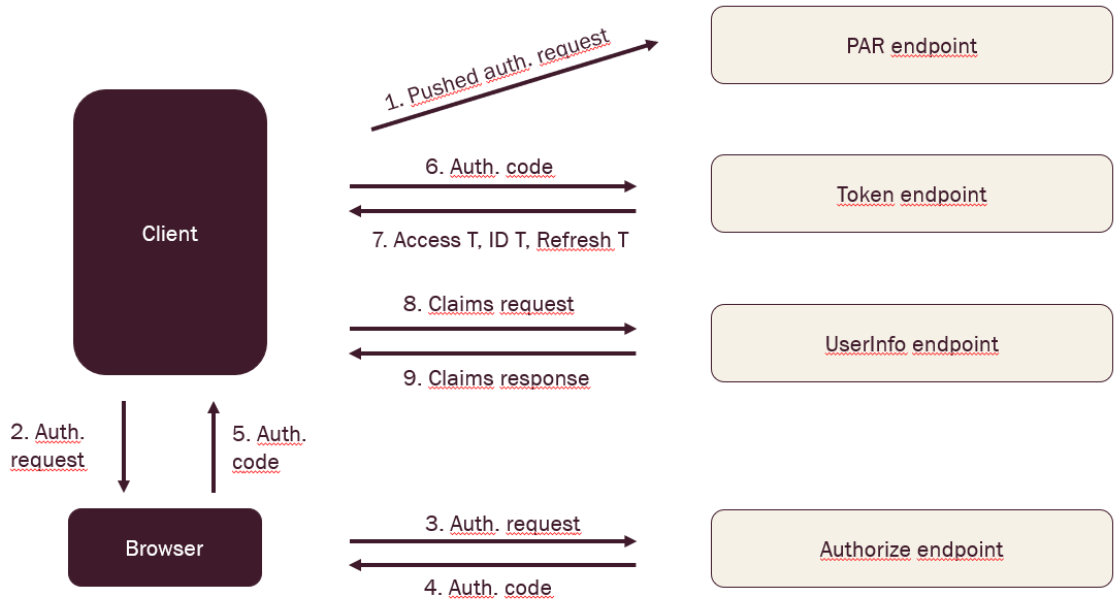


Figure 2: Authorization flow

The main steps of the flow are:

1. The client pushes an authorization request and authenticates to the /par endpoint (if supported by the client type, e.g. confidential client). If pushed requests are not supported by the client type, the request parameters are sent in step 2.
2. The client opens/redirects a browser with an authorization[6] request or reference to previously pushed request.
3. The Authorization endpoint receives the authorization request, authenticates the user, and obtains end-user consent for the requested scopes. Authenticating the user may involve chaining to other authentication systems.
4. The Authorization server issues an authorization code and returns it to the redirect URI.

[6] Note that in [OIDC] this is called an authentication request, whereas OAuth calls it an authorization request.

5. The client receives the authorization code from the redirect URI.
6. The client presents the authorization code at the token endpoint.
7. The token endpoint validates the authorization code and issues the tokens requested: Access Token, ID Token and Refresh Token. The client validates the response (not shown on figure above).
8. The client optionally requests further user attributes/claims from the UserInfo endpoint using the Access Token obtained previously.
9. The Authorization Server returns a User Token with the requested claims.

4.1 Authorization requests

[OIDC-01]

The client authorization protocol MUST follow the [OAuth] 2.0 authorization code grant type as defined in section 4.1 of [OAuth]. Unless otherwise stated explicitly, the requirements from this specification apply directly.

[OIDC-02]

The client SHOULD use pushed authorization requests defined in [PAR] with client authentication if supported by the client type (e.g. confidential clients). Client authentication SHOULD use the private_key_jwt method defined in [OAuth] and [OIDC]. A response to a pushed request MUST only be valid for 600 seconds.

[OIDC-03]

The request parameters in the authorization request MUST follow the requirements specified in the table below:

Parameter	Mandatory	Usage
scope	Y	MUST contain the 'openid' scope value and MAY contain scopes for external APIs which the user should authorize. See section 3.2 for a description of the authorization model. The client MAY specify one or more of the following scope values representing the client's desired attribute profiles defined in [OIO JWT]: person_dk person_dk_withoutcpr person_dk_anonymous professional_dk professional_dk_anonymous person_eu legalperson_eu professional_eu The client MAY further specify scopes for individual attributes defined in the above attribute profiles in [OIOJWT].
response_type	Y	MUST be set to 'code'
client_id	Y	MUST be set to the client identifier (Entity ID) pre-registered with the Authorization Server.

Parameter	Mandatory	Usage
redirect_uri	Y	The client MUST use a claimed HTTPS redirect URI when supported by the client platform[7] (e.g. “https://app.example.com/oauth2redirect/example-provider”) and the URI MUST NOT contain any wildcards. This ensures the identity of the destination client to the authorization server by the operating system. If the mechanism is not supported, the client SHOULD instead use a “custom URL scheme” for URI redirection, and it MUST be URI scheme based on a domain name under control of the client developer as described in RFC7595. It is REQUIRED that a unique redirect URI is used for each authorization server used by the client.
state	Y	To mitigate CSRF-style attacks over inter-app URI communication channels (so called ‘cross-app request forgery’), it is REQUIRED that the client includes a high-entropy secure random number (>=128 bit) in the "state" parameter of the authorization request.
code_challenge	Y	The client MUST use the Proof Key for Code Exchange ([PKCE], RFC7636) extension to OAuth and include a code challenge derived from high-entropy cryptographic random valued.
code_ challenge_method	Y	MUST be ‘S256’ (see [PKCE]).
nonce	Y	MUST include a high-entropy secure random number (>=128 bit) in order to prevent ID token replay.
acr_values	N	String that specifies the acr value that the Authorization Server is being requested to use for processing this Authentication Request. In this profile the following generic LoA levels are used which can be mapped to either NSIS or eIDAS assurance levels: https://data.gov.dk/concept/core/loa/Low https://data.gov.dk/concept/core/loa/Substantial https://data.gov.dk/concept/core/loa/High The Authorization Server MUST ensure that the end-user is authenticated at least to the specified LoA level or return an error.
prompt	N	The parameter prompt=none MAY be used to request passive authentication where no user dialog is presented. This MAY however limit the scopes which can be requested. The parameter prompt=login MAY be used to request a forced authentication.

[7] This is supported both on iOS and Android 6.0 and above.

Note: PKCE is a proof-of-possession extension to OAuth 2.0 that protects the authorization code from being used if it is intercepted. The extension has the client generate a secret verifier; it passes a hash of this verifier in the initial authorization request and must present the un-hashed verifier when redeeming the authorization code. An attacker that intercepted the authorization code would not be in possession of this secret, rendering the code useless.

The following authorization request parameters SHOULD NOT be used with this profile: `display`, `response_mode` and `id_token_hint`.

Other request parameters defined in [OIDC] and [OAuth] and not mentioned here are all OPTIONAL.

[OIDC-04]

> OAuth 2.0 authorization requests from the client MUST be sent through external user agents (i.e. not embedded web views in a native app). Otherwise, a client being a native app may be able to copy user credentials and cookies. In-app browser tabs MAY be used if they separate security context from the native app.

Note: the above requirement may imply that users have an authenticated session in their browser with an IdP, even after the app has been closed. If supported, the client SHOULD instruct the IdP to avoid session establishment in the browser if the client is a native app.

4.2 Authorization response

[OIDC-05]

> The Authorization Server MUST validate the request as specified in section 3.1.2.2 (Authentication Request Validation) of [OIDC] including that all required parameters mentioned above (section 4.1) are present. Hence, the `scope` parameter MUST contain the `openid` scope value.
>
> As specified in [OAuth], Authorization Servers SHOULD ignore unrecognized request parameters.

[OIDC-06]

> If the client is a confidential client, the authentication data sent by the client (e.g. signature) MUST be validated against the registered credential (e.g. OCES certificate).

[OIDC-07]

> The Authorization Server MUST reject a `redirect_uri` in requests that doesn't *exactly* match the one that was previously registered.

Note: As mentioned under prerequisites, the client must register its redirect URI with the Authorization Server and it must be unique per Authorization server.

[OIDC-08]

The Authorization Server MUST record the [PKCE] challenge and method in the request and reject requests not containing these parameters.

[OIDC-09]

If the request specifies a desired level of assurance (LoA) it must be honored by the Authorization Server. If the LoA cannot be met, an error response MUST be sent. If the client type is a native app, the authentication MUST be a fresh authentication of the end-user (e.g. SSO is not permitted here).

[OIDC-10]

After successful authentication of the end-user, the Authorization Server MUST obtain (and securely store) user consent for any user-granted scopes defined in the request. Note that attribute profile scopes listed in [OIDC-3] are not considered user-granted scopes and do not trigger consent. Individual attributes (e.g. CPR number) MAY trigger consent based on local policy requirements.

The user SHOULD be able to give consent individually per scope in the request.

[OIDC-11]

After successful authentication of the end-user and granted consent, the Authorization Server MUST issue an authorization code, and the Authorization Response MUST return the parameters defined in Section 4.1.2 of [OAuth] by adding them as query parameters to the `redirect_uri` specified in the Authorization Request using the `application/x-www-form-urlencoded` format.

For web clients, the Authorization Server MUST include a `session_state` parameter in order to enable session management (see chapter 5) as described in the OIDC Session Management Specification.

4.3 Client receives the authorization code

[OIDC-12]

The client MUST validate the response according to [OAuth] especially Sections 4.1.2 and 10.12.

[OIDC-13]

The client MUST validate `state` in responses and MUST reject responses if they do not match a pending outgoing authorization request. The client MUST further compare the redirect URI in the response to the value used in the authorization request and MUST verify that the URI on which the authorization response was received exactly matches it.

4.4 Code exchange

[OIDC-14]

> The client MUST send a token request to the Token endpoint to obtain a token response as described in Section 3.2 of [OAuth], using the `grant_type` value `authorization_code`.
>
> Confidential clients MUST authenticate using the private_key_jwt method defined in [OAuth] and [OIDC].

[OIDC-15]

> The client MUST include the PKCE `code_verifier` secret matching the `code_challenge` sent in step 1.

4.5 Token endpoint validates the authorization code and issues the tokens requested

[OIDC-16]

The Token endpoint MUST validate the Token Request as described in OpenID Connect Core section 3.1.3.2 including the presented authorization code, PKCE `code_verifier` and value of `redirect_uri` parameter.

[OIDC-17]

The Token endpoint MUST issue an ID Token, an Access Token and MAY issue a Refresh Token according to section 3.1.3.3 of OpenID Connect Core.

[OIDC-18]

The ID Token MUST be a JWT token according to the OIO JWT Profile [OIO JWT] and include an `at_hash` claim[8]. The Access Token and Refresh Tokens MAY be opaque and SHOULD include at least 128 bit of entropy.

Generally, ID Tokens SHOULD be restricted to information regarding the authentication event.

The Access Token is targeted for the internal Authorization Server endpoints and MAY be opaque.

Access Tokens MAY be sender-constrained either by including a cnf claim in a JWT token, or by using introspection with state stored server-side with an opaque token.

4.6 Client validates response

[OIDC-19]

The client MUST validate the response according to section 3.1.3.5 (Token Response Validation) of OIDC Core. In addition, the `at_hash` value MUST be validated as specified in section 3.1.3.6 of OIDC to ensure binding between Access Token and ID Token.

[OIDC-20]

[8] When an OIDC provider issues an ID token and an access token together, it includes the at_hash claim inside the ID token as a cryptographic proof that both tokens belong to the same authentication response.

The client MUST validate that the signature of the ID Token is valid, that it uses an allowed signing algorithm defined in the OIO JWT Profile [OIO JWT], using a pinned[9] token-signing certificate of the Authorization Server. It MUST also verify, that it is the audience (`aud` claim) of the token, and the value of the `nonce` element.

The client MUST further check that the resulting assurance level in the ID Token lives up to its requirements as well as other claims in the ID token required by the client.

4.7 Client requests additional attributes

[OIDC-21]

The client MAY invoke the UserInfo endpoint at the Authorization Server to request additional claims/attributes about the user.

The response MUST only contain claims whose corresponding scopes have been consented by the user during authorization (e.g. user granted scopes). Other claims not requiring user consent MAY be included in the response if allowed by the Authorization Server depending on client type and local policy[10].

The returned token MUST be a signed JWT token according to the OIO JWT Profile [OIO JWT].

[9] E.g. part of the App configuration.

[10] For example sensitive claims such as CPR may be restricted to certain client organisations.

5. Token renewal and session management

Refresh Tokens are credentials used to obtain Access Tokens. Refresh tokens are issued to the client by the Authorization Server and are used to obtain a new Access Token when the current Access Token becomes invalid or expires.

The profiles described in this document rely on the principle that issued Access and Delegated Access Tokens are relatively short-lived (e.g. one hour or less) such that they have to be refreshed often. This approach has several benefits:

- A short validity period reduces attack windows.
- Continuously refreshing Access Tokens means that their content can/will be updated – e.g. if the user has withdrawn their consent to an app or if the app has been revoked.
- Recipients of Access Tokens (e.g. API providers) are not burdened with having to check for token revocation by calling external services.
- Token revocation functionality can be focused on Refresh Tokens, which have a potentially longer validity period. It can be handled internally in the Authorization and Token Servers.

When all Access Tokens and the associated Refresh Tokens have expired, the client has to obtain new tokens using the Client Authorization Profile described in chapter 4. User interaction can be avoided if the Refresh Token remains valid.

Note: In an attempt to protect users from excessive tracking and surveillance, major browser vendors have introduced increasingly restrictive anti-tracking measures including Chrome's SameSite and Safari's ITP2. These mechanisms may impact identity and therefore must be considered in real-world deployments. As a result, most of the requirements in this section are given as recommendations (SHOULD) to provide flexibility to maneuver around these challenges.

5.1 Using a Refresh Token

A client with a valid Refresh Token can use it to obtain a new Access Token for the Token Server and then use this Access Token to contact the UserInfo endpoint or obtain new Delegated Access Tokens for SP APIs.

Refresh tokens are used with the `refresh_token` grant type as described in section 12 of [OIDC]; below is shown an example:

```
POST /token HTTP/1.1
  Host: server.example.com
  Content-Type: application/x-www-form-urlencoded

  client_id=s6BhdRkqt3
  &grant_type=refresh_token
  &refresh_token=8xLOxBtZp8
  &scope=openid%20profile
```

5.2 Token revocation

[OIDC-51]

> Authorization Servers that issue long-lived (>=8 hours) Refresh Tokens MUST provide a mechanism to revoke those tokens including a token revocation endpoint compliant with [RFC7009]. Thus, tokens can be revoked by making an HTTP POST request to the revocation endpoint as specified in this RFC.

This profile does not define the specific circumstances or policies where Refresh Tokens have to be revoked – only the capability to revoke them is required. Examples of circumstances that could lead to revocation in specific implementations are:

- A user interface could be provided to the end-users allowing them to revoke Refresh tokens for individual App instances or all instances running on a particular device (e.g. when the user has lost a device). This might also be used by support personnel in case the end-user has temporarily lost their ability to authenticate (e.g. because their credential is on the same device).
- Refresh Tokens that are not frequently used could be set to be revoked automatically (e.g. 3 months of inactivity).
- Clients which are native apps can be programmed to request revocation when certain criteria are met on the end-user device – for example if the user has changed biometry on their phone, if a wrong user-pin is entered a certain number of times, or an indication of compromise is detected.
- External events (e.g. the user revoking their user credential such as NemID or MitID) could be configured to automatically revoke Refresh Tokens enrolled and authorized with the credential – provided that these events can be propagated. Other events could be that the user identity is revoked (e.g. an employee identity no longer being associated with a company), or an API provider revoking all access to a client.

Revocation policies are internal to the implementation of the Authorization Server and do not affect the wire protocol; they are therefore left to implementations to decide.

5.3 Refresh token rotation

[OIDC-52]

> For clients that are JavaScript applications with no backend (i.e. SPAs), the Authorization Server MUST rotate Refresh Tokens on each use and ensure mechanisms to detect token replay (as described in [OSBP] section 4.12). Further, the lifetime of the new refresh token MUST NOT extend the lifetime of the initial refresh token.

5.4 Token lifetime

[OIDC-53]

The lifetime of tokens SHOULD be limited.

Deployments are recommended to use the values in the table below:

Client type	Native app	Web/JS app with a Backend	JS app without a Backend
Refresh Token	Long-lived refresh tokens allowed (not expiring) if revocation mechanism implemented and token rotation implemented	8 hours	1 hour (with rotation)
ID Token	1 hour	1 hour	1 hour
Access Tokens	1 hour	1 hour	1 hour
User Tokens	1 hour	1 hour	1 hour

5.5 Session management

Clients that are native apps are not considered to have a (web) session based on the initial end-user authentication. For web clients, the session management requirements are stated below based on the client type.

5.5.1 Session management for web apps with backend

[OIDC-54]

> Web clients with a backend SHOULD be able to receive logout events from the Authorization Server using [OIDC] Front Channel Logout or [OIDC] Back Channel Logout, as supported by the Authorization Server.

When terminating a session, in response to a logout request, both session cookies and tokens MUST be discarded by the app backend.

[OIDC-55]

The Authorization Server MUST propagate logout events to/from any federated authentication server (i.e. SAML IdP) used to authenticate the end-user in addition to own relying parties involved in current session.

5.5.2 Session management for web apps without backend

[OIDC-56]

Web clients with no backend (i.e. SPAs) SHOULD continuously[11] poll the Authorization Server for changes to the user session via the hidden iframe mechanism defined in the OIDC Session Management Specification.

When a session change is detected, all tokens and browser local storage MUST be discarded by the app. Local storage SHOULD be avoided by SPAs for sensitive tokens where possible.

5.5.3 Client-initiated logout

The previous sections cover logout events occurring outside the client. Web clients however also need to be able to inform the Authorization Server that the end-user has requested (single) logout.

[OIDC-57]

Authorization Servers MUST implement a Logout endpoint according to the OpenID Connect RP-initiated logout protocol [LOT] which can be used by clients to initiate logout by redirecting the user agent.

In this profile, the `id_token_hint` parameter MUST be sent by the client and contain an ID Token previously issued by the Authorization Server.

[OIDC-58]

As part of logging out the end-user, the Authorization Server MUST use relevant logout mechanisms registered by (other) clients (as specified in sections 5.5.1 and 5.5.2) to notify that they are to likewise log out the end-user.

[11] Every 10 seconds or less. As mentioned in the introduction, this may be inhibited by some browsers (SameSite etc.).

6. Token exchange profile

This chapter describes a token exchange profile based on OAuth Token Exchange [OTE]. In the profile, the client requests an Access Token not for the Authorization Server but for an external API on behalf of the user. This token is called a Delegated Access Token[12] to separate it from the internal Access Token issued for the OIDC Authorization Server. The Delegated Access Token is targeted for a specific SP API (via an aud claim) and includes scopes covering this API.

The client requests the token by authenticating with a client credential and passing the Access Token issued during the authorization phase as a parameter. This Access Token represents the scopes and permissions granted by the end-user to the client.

The Token Server verifies that the presented Access Token and credential are valid, and that the end-user has previously authorized the client instance to use any user-granted scopes (via the consent gathered in step 3 of the Client Authorization Profile). If the request is successful, the Token Server issues a new Delegated Access Token for the API according to the [OIO JWT] Profile, and the scopes for the API are encoded as privileges in a JSON structure. See also section 3.2 for further detail.

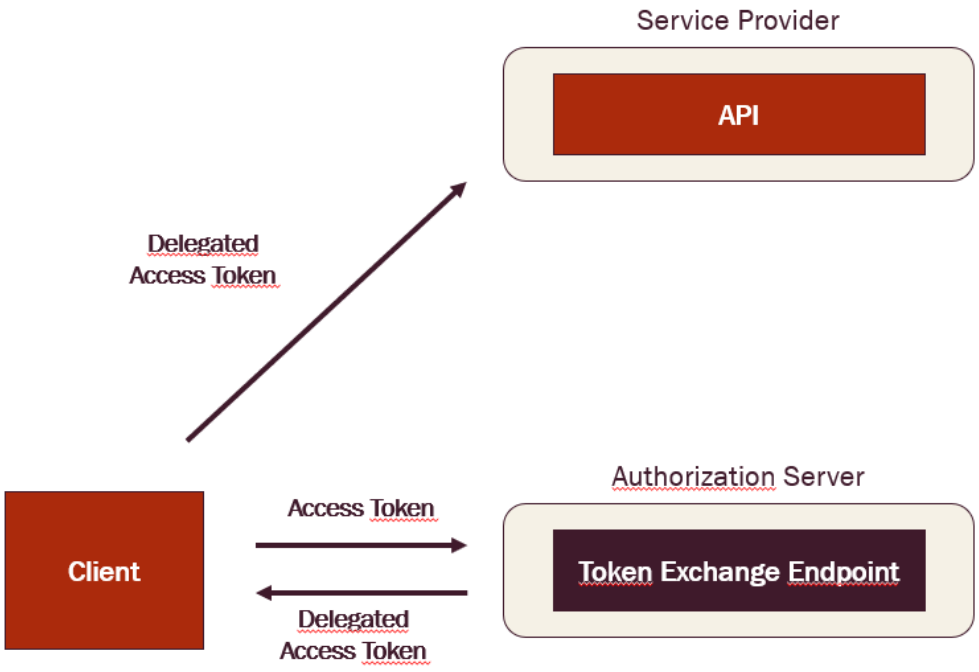


Figure 3: Token Exchange Scenario

6.1 Client sends Token Request

[OIDC-61]

The token request MUST use the OAuth token exchange [OTE] protocol with client authentication using the private_key_jwt method.

[OIDC-62]

The request MAY contain a DPoP header with a signed proof JWT. If a DPoP proof is present in the request and valid the issued token MUST be DPoP-bound. If it is invalid, the Token Server should reject the request with an error.

[OIDC-63]

The request parameters in the token request MUST fulfill the requirements specified in the table below:

Parameter	Mandatory	Usage
subject_token	Y	Value MUST be an access token representing authorization granted by the end-user to the client.
subject_token_type	Y	MUST be `urn:ietf:params:oauth:token-type:access_token.`
grant_type	Y	Value MUST be `urn:ietf:params:oauth:grant-type:token-exchange`
audience	Y	Value MUST be the EntityID of the external API.
scope	Y	Value MUST contain list of scope values belonging to at most one external SP API registered with the Authorization Server.
requested_token_type	Y	Value MUST be `urn:ietf:params:oauth:token-type:access_token`

Note: the subject_token is chosen to be the access token and not the ID token from the authorization phase since the access token represents the delegated authorization – i.e. proves that the user authorized the client to act on the user's behalf within a certain scope. Note also that the client_id parameter can be omitted since the client identity is conveyed inside the authenticating JWT itself via the sub claim.

A sample Token Request is shown below:

```
POST /token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
DPoP: <signed-dpop-proof-jwt>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange
&client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer
&client_assertion=<signed-jwt>
&audience=https://provider.dk/service_xyz
&scope=read_profile
&subject_token=<access_token>
&subject_token_type=urn:ietf:params:oauth:token-type:access_token
&requested_token_type=urn:ietf:params:oauth:token-type:access_token
```

6.2 Step 2: Token Server validates request and returns token

[OIDC-64]

The Token Server MUST verify the client authentication and that requested `scope` values for user granted scopes have previously been consented by the end-user for the particular client.

[OIDC-65]

If the request is successful and authorized, the Token Server MUST issue a new Delegated Access Token according to [OIO JWT] Profile with the requested scopes converted to privileges. This includes both user-granted scopes and Service Provider granted scopes (see section 3.2 for details.)

[OIDC-66]

The issued token SHOULD have a limited validity period.

[OIDC-67]

When the client sends a DPoP proof to the token endpoint, the Token Server MUST validate it and compute cnf.jkt from the proof key.

[OIDC-68]

The resulting Delegated Access Token MUST identify the end-user as the subject and the client as the actor, as specified in the [OIO JWT] Token Profile.

7. API Access Profile

This chapter describes how a client can invoke an external SP API using a Delegated Access Token obtained via the mechanisms described in the Token Request profile in chapter 6.

[OIDC-71]

> The client MUST pass the Delegated Access Token in an `Authorization` HTTP header using the `Bearer` scheme, if the token is not sender-constrained.

Example using Bearer token:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer <access_token>
```

[OIDC-72]

> If the Delegated Access Token is sender-constrained, the client MUST pass the token as type DPoP and generate an associated request-specific DPoP proof JWT.

Example with DPoP token:

```
GET /resource HTTP/1.1
Host: external-api.example.com
Authorization: DPoP <access-token>
DPoP: <signed-dpop-proof-jwt>
```

[OIDC-73]

> The SP API MUST validate the received Delegated Access Token including (as a minimum) that it is not expired, that the signature is valid, that it is signed by a trusted Token Server using an allowed algorithm, that the SP API is the intended audience of the token (`aud` claim), and that required privileges are included (`priv` claim). See the [OIO JWT] profile for details.

[OIDC-74]

> The SP API MUST validate that assurance level (for the end-user authentication) asserted in the Access Token (`loa, nsis_loa, eidas_loa etc.`) is sufficient according to local access policy. The SP API MAY also consider the authentication time of the end-user, (`auth_time` field) before access is granted.

[OIDC-75]

For a DPoP-bound Delegated Access Token, the SP API MUST validate the access token, extract cnf.jkt, validate the DPoP proof JWT signature and claims, verify that htm and htu match the request, verify ath against the presented access token, verify iat freshness, and reject replayed jti values within the configured replay window

8. Security Requirements

The generic security requirements below apply to all protocol profiles in this document. This is a minimum baseline and implementors of clients and APIs should consider additional implementation-specific security requirements according to a risk assessment.

[OIDC-81]

All transport communication between the client and the authorization infrastructure MUST use TLS 1.2 or higher and SHOULD only use cipher suites supporting perfect forward secrecy. Servers MUST reject negotiation of insecure TLS connections. The document [NIST 800-52] (section “Minimum Requirements for TLS Servers”) or subsequent revision may serve as reference for an acceptable level of transport security.

[OIDC-82]

Native app clients MUST pin server TLS certificates (i.e. maintain a list of trusted TLS server certificates as part of their configuration).

[OIDC-83]

JWT tokens MUST follow the OIO JWT Profile [OIO JWT].

9. References

[JWA] Jones, M., “JSON Web Algorithms (JWA), IETF Proposed Standard", RFC7518, May 2015.
https://datatracker.ietf.org/doc/html/rfc7518

[JWE] Jones, M., and J. Hildebrand, “JSON Web Encryption (JWE), IETF Pro-posed Standard”
https://tools.ietf.org/html/rfc7516

[JWK] Jones, M., “JSON Web Key (JWK),” IETF Proposed Standard,
https://tools.ietf.org/html/rfc7517

[JWS] Jones, M., Bradley, J., and N. Sakimura, “JSON Web Signature (JWS),” IETF Proposed Standard,
https://tools.ietf.org/html/rfc7515

[JWT] Jones, M., Bradley, J., and N. Sakimura, “JSON Web Token (JWT),” IETF Proposed Standard,
https://tools.ietf.org/html/rfc7519

[OIO JWT] “OIO JWT Token Profile 1.0”, Danish Agency for Digital Government.
https://digst.dk/openid-connect/

[HOK] Campbell, Bradley, Sakimura: "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens",
https://datatracker.ietf.org/doc/html/rfc8705

[LOT] Jones, Medeiros, Agarwal, Sakimura, Bradley: "OpenID Connect RP-Initiated Logout 1.0",
https://openid.net/specs/openid-connect-rpinitiated-1_0.html

[NSIS] ”National Standard for Identiteters Sikringsniveauer 2.1.0”.
https://digst.dk/nsis

[OIOSAML] ”OIOSAML Web SSO Profile 4”.
https://digst.dk/OIOSAML/

[OIO-BPP] ”OIO Basic Privilege Profile 1.2”.
https://digst.dk/oioidws

[RFC8252] “OAuth 2.0 for Native Apps”, IETF.

[RFC6750] “The OAuth 2.0 Authorization Framework: Bearer Token Usage”, IETF,
https://tools.ietf.org/html/rfc6750

[RFC7009] ”OAuth 2.0 Token Revocation”, IETF.

[OIDC] “OpenID Connect Core 1.0 incorporating errata set 2, November 2014”, OpenID.Net.

[OAuth] “The OAuth 2.0 Authorization Framework”, RFC6749, IETF, October 2012.

[OSBP] ”Best Current Practice for OAuth 2.0 Security”, IETF.
https://www.rfc-editor.org/rfc/rfc9700.html

[DPoP] “RFC 9449 OAuth 2.0 Demonstrating Proof of Possession (DPoP)”,
https://www.rfc-editor.org/rfc/rfc9449.html

[PAR] “OAuth 2.0 Pushed Authorization Requests”,
https://datatracker.ietf.org/doc/html/rfc9126

[OTE] “OAuth 2.0 Token Exchange”,
https://datatracker.ietf.org/doc/html/rfc8693

Agency for Digital Government