Best practices for integrating OpenID Connect / OAuth2 based end services

TOC (gray indicates done)

- Endpoints and IDtoken:
    - What should be available at which endpoint
    - Userinfo vs. introspection endpoint [RFC7662] (e.g. non-authenticated vs authenticated etc.).
    - List of endpoints (maybe a table)
- Dynamic registration, public clients, Registration with AP => Davides Document!!!, PKCE
    - Consider not to allow token exchange for public clients (i.e. anybody with the (public) client_id and an AT can exchange it into an refresh token. This may open the door for malicious behaviour just a bit too much.
    - Restrict registration of confidential clients to specific users/groups for the token exchange grant type.

- Claims
    - How to request things:
        - Note: Reference G026 regarding to put the Community identifier into the ISS claim (Max leng of iss is 255, though, scope is also 255)!!! (see https://openid.net/specs/openid-connect-core-1_0.html#SubjectIDTypes) ePUID@SCOPE should not exceed 255 (this needs to go into G026) What about the “@” sign?
        - Scopes vs. Claims
    - R&S
        - Explicitly write the R&S attributes in OIDCRE notation
        - Expect the OIDCre whitepaper attributes
    - Assurance
        - [MS]FA -> acr claim
        - RAF statements -> eduperson_assurance
        - Assurance Profiles (RAF, AARC, IGTF, ...)
    - Group membership and role information
        - AARC-G002 (just examples)
    - Capabilities
        - AARC-G027 (Examples)
    - Affiliation
        - AARC-G025

Abstract

This document provides information about how the information should be exchanged in FIM scenarios for OIDC/OAuth2 services. It will summarise the current best practices regarding which features should be supported. It will also consider flows and standards that may influence how the services, i.e. clients, operate. At best, it will provide recommendations.

Introduction

This document covers several aspects. First we will describe the targeted use-cases, which are web, command-line, and API based. Then we will describe the different flows and tokens and give recommendations on when to apply and how to use them. Finally, we will cover the available claims and how they correspond to attributes in the SAML domain.

OAuth2/OIDC disclaimer

With this document we provide practical guidance for integrating OpenID Connect (OIDC) based services. OIDC builds on top of the OAuth2.0 [RFC6749] standard. For the sake of simplicity we do not always distinguish clearly between the two.

OIDC is often described as the identity layer on top of the authorisation layer that OAuth2 provides[1]. For example, OAuth2 can be used for the use case in which a user authorises a printing service to access photos (the protected resource) on a google drive (this is the authorisation “scenario”, i.e. user authorises an “action”). In this scenario, the printing service requests an Access Token (AT) from the google authorisation server (AS , or OpenID Connect Provider OP). The google AS asks the user for his consent.

In the scientific infrastructure scenario, OAuth2 and OIDC are used in a similar, but different way: The end service (relying party, RP) needs to understand who a given user is to grant or deny access. For this, it uses the AT to access to userinfo-endpoint. Information returned are used for an authorisation decision. Other than in OAuth2, In OIDC OP and AS reside on the same instance.

Effectively, ATs are passed around with jobs (i.e. remote processes running on behalf of a user). They are verified by trying to access the userinfo-endpoint. This is probably not exactly how the original specs were meant to be used.

[1] https://openid.net/connect/

Relevant use-cases

This document is written with the background that the following use-cases should be covered:

Web

In this case, the end user is using its web-browser directly to access a service. If not already logged in, the user will be redirected to an OpenID Connect Provider (OP, equivalent to IdP in SAML) for authentication. After successful authentication, the user is presented with information that is passed on to the service. The SAML analogy holds also for Single Sign On (SSO): Cookies in the browser will make sure that subsequent authentications work, during a reasonable session lifetime, without prompting the user. In OIDC and OAuth2 this is usually achieved using Authorisation Code Flow.

Commandline

In this case, a client application (usually CLI application) needs to obtain an access token. The challenge is that most flows are targeted at web browsers. Oauth2 defines several flows for the commandline. Combinations of commandline (CLI) and web flows are possible.

API calls

REST API calls may be authenticated with OAuth2/OIDC. The standard pattern is to pass the AT as a Bearer Token in in the “Authorisation header”:

```
curl https://example.com/api -H "Authorisation: Bearer $AT”
```

The server of the REST API needs to verify the received AT with the OP. Offline verification may be possible, if the AT is a signed JWT.

Multiple OIDC Providers

Additional challenges arise in cases where multiple OPs exist. Such cases are currently only supported in SAML and federations, where many different IdPs are trusted by services. In the OIDC specification this in not defined yet. This is addressed within the OpenID Foundation by a working group on OIDC federations [OIDF], but it is work in progress. The proxy concept reduces the urgency of the need to some extent.
As a consequence of this, it is not possible to tell from an AT, which OP has issued it.
This is addressed at some OPs by using JSON Web Tokens (JWTs) [RFC7519]. This allows passing (among other information) the OP’s URI in the issuer (“iss”) claim.

OIDC Overview

This section provides down-to-earth information about the different token types, how to obtain tokens, how to verify them, and guidance on understanding which information may be obtained using which token. References to later sections (or the appendix) will contain the lengthy information.

Token Types

Several tokens are defined. All tokens are issued by the AS / OP.

- Access Token (AT) (OAuth2/OIDC): Typically an opaque string, but it may be a JWT[2]. Allows accessing the userinfo endpoint at the OP, i.e. obtaining information about the user (as in attributes, entitlements, name, ...). ATs are not bounded, i.e. they may be used without client_id/client_secret. ATs are short lived. We see times between 300 and 3600 s.
  Some implementations encode userinfo-endpoint information (or a subset thereof) inside the AT, using JWT. This is done to address scalability concerns, because otherwise each AT has to be verified with a call to the OP.
- ID Token (OIDC): The ID token is a JWT that contains identification information. This is typically information about the identity of the user (for example assurance levels, requested scopes, ...). ID token are given to clients as part of authentication flows.
- Refresh Tokens (RT) (OAuth2, OIDC): Refresh tokens are long lived tokens that allow requesting new ATs. RTs may only be used by the client_id for which they were issued.

To obtain attributes (claims in OIDC), clients need to request so called scopes. Scopes contain sets of claims. The list of supported scopes is advertised by the OP in his configuration endpoint at "`/.well-known/openid-configuration`".
The actual information contained in a token, respectively the information accessible with a given token is not specified in OIDC.

Currently, no common standard has formed that describes which information should be available via which token or in which way. The reader is advised to carefully compare best practices across different OP implementations.

OIDC Flows

"OpenID Connect is a is a simple identity layer on top of the OAuth 2.0 protocol."[OIDC-CORE] Therefore, flows or grants supported by Oauth2 may be also supported by OIDC. In the Figure we show the general OIDC Protocol diagram. A selection of flows is shown in Appendix A.

[2] https://tools.ietf.org/html/rfc7519

```
+--------+                                   +--------+
|        |                                   |        |
|        |---------(1) AuthN Request-------->|        |
|        |                                   |        |
|        |  +--------+                       |        |
|        |  |        |                       |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                       |        |
|   RP   |  |        |                       |   OP   |
|        |  +--------+                       |        |
|        |                                   |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                   |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                   |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                   |        |
+--------+                                   +--------+
```

OIDC Protocol diagram (in abstract)

Steps in the the OIDC protocol (in abstract):

1. The RP (Client) sends a request to the OpenID Provider (OP).
2. The OP authenticates the End-User and obtains authorisation.
3. The OP responds with an ID Token and usually an Access Token.
4. The RP can send a request with the Access Token to the UserInfo Endpoint.
5. The UserInfo Endpoint returns Claims about the End-User.

Endpoints

In OAuth2, there are at several endpoints (as defined in the spec [RFC-6749]):

- Authorisation endpoint - used by the client to obtain authorisation from the resource owner via user-agent redirection.
- Token endpoint - used by the client to exchange an authorisation grant for an access token, typically with client authentication.
- Redirection endpoint - used by the authorisation server to return responses containing authorisation credentials to the client via the resource owner user-agent.
- Introspection Endpoint[3] - This endpoint is an OAuth2 endpoint that takes a parameter representing a token and returns a JWT containing the information about the token,

[3] https://tools.ietf.org/html/rfc7662

including whether the token is still active. It may also include more information, such as "scope" list associated with the token, "token_type", etc.

OIDC defines an additional endpoint:

- UserInfo Endpoint[4] - an OAuth2 Protected Resource that returns claims about the authenticated End-User.

In OIDC, when requesting information (scopes and claims) about the users, the information may be contained in the ID token, or it may be in the obtained from the UserInfo Endpoint. If no Access Token is issued (as when "response_type" parameter has value "id_token"), then the resulting claims are contained in the ID token. However, when an Access Token is issued (e.g. when using "response_type=code") then the claims obtained when requesting scopes (using "scope" parameter) such as "profile", "email", "address", "phone" (which are scopes defined in the OIDC spec[5]) or other defined scopes are obtained from the UserInfo Endpoint. Claims can also be explicitly requested, using "claims" parameter.

Endpoint	OIDC (Authorisation Grant)	OAuth2	Authorisation required for access
Authorization	Authentication and Authorisation of the user	Same	Not specified (this is where the user authenticates)
Token	Obtain Access Token, ID Token (optionally Refresh Token)	Same, without ID Token	authorization grant or refresh token
UserInfo	Obtaining Claims (using "scope" or "claim" parameters)	Not defined	Access Token
Introspection	?	Obtain information about the Token (e.g. "scope" supported)	Client_id + client_secret
Redirection	The client URI where the user agent is redirected to	Same	
Token Exchange			Access Token / Refresh Token

[4] https://openid.net/specs/openid-connect-core-1_0.html#UserInfo
[5] https://openid.net/specs/openid-connect-core-1_0.html#Claims

Claims

Claims in OIDC correspond to attributes and entitlements in SAML. They express what the OP "claims" to know about a given subject. The difference to SAML is that claims can be grouped into scopes. Most scopes are not defined by the OIDC Specification. A list of supported scopes may be provided by the OP by default. A client may request a list of scopes. For practical applications clients SHOULD[6] specify which scopes they require.

Requesting claims

Data minimisation and privacy by design are dictating that the services should only request the information they need to properly function. This, however, should not be interpreted that they should ask for as little as possible personal information, but only what is necessary. As defined in Data Protection Code of Conduct[7], enabling access to a service covers, among others, Authorization, Identification, Accounting and Billing, Information Security. Therefore, access to a service may require comprehensive information, and that is in line with the principles outlined in the GDPR. Naturally, other technological and organisational measures may be required, but this is out of scope of this document.

For the basic profile, profile and email can be requested and this provides a consistent set of attributes. Being based on R&S[8], it is already following data minimisation guidelines, since further minimisation of information does not provide additional benefits, but may however impact the operation of services.

For the advanced profile, OIDC allows for requesting individual claims or non-standard scopes.

Mapping SAML attributes to OIDC Claims

The "White Paper for implementation of mappings between SAML 2.0 and OpenID Connect in Research and Education"[9] considers the mapping between the SAML attributes and OIDC scopes/claims. It defines a basic and an advanced profile. For the basic profile, recommendations is shown in the table below:

OIDC Scope	**OIDC claim**	**eduPerson attribute**

[6] GDPR mandates that no more information than those required to provide a service MUST be requested.
[7] https://wiki.refeds.org/display/CODE/Data+Protection+Code+of+Conduct+Home
[8] https://refeds.org/category/research-and-scholarship
[9] https://wiki.refeds.org/display/CON/Consultation%3A+SAML2+and+OIDC+Mappings

profile	Public sub	eduPersonPrincipalName (if non-reassigned) or eduPersonTargetedID
	name	displayName
	given_name	givenName
	family_name	sn (surname)
email	email	mail[10]
	email_verified	See [OIDCRE-WHITE]
eduperson_entitilement	eduperson_entitlement	eduPersonEntitlement (see [G002] & [G027])
openid?	acr	AuthenticationContextClassRef (see [SAML-CORE-SPEC], [SFA] & [MFA])
eduperson_assurance/openid?	eduperson_assurance	eduPersonAssurance (see [RAF] & [eduPerson])
eduperson_scoped_affiliation	eduperson_scoped_affiliation	eduPersonScopedAffiliation (see G025 & [eduPerson])
voperson_external_affiliation	voperson_external_affiliation	voPersonExternalAffiliation (see G025 & [voPerson])

It is recommended to set the email_verified claim to "true" if the email address that is being provided in the claim was:

- Provided by the Institutional Identity Provider as part of the SAML assertion, and
- The domain part of the email address is a (sub) domain of the institution
- The domain of the email is validated by the implementation based on the <shibmd:Scope> element from the entities SAML metadata.

As in such case it may be assumed the email service being used is under direct administrative control of the Institution, and the requirements for setting email_verified to "True" have been fulfilled.

[10] As mail may be multi valued, it is left to the implementer to choose which address needs to go into the single valued email claim

For the advanced profile, and further attribute mappings, going from SAML to OIDC, the white paper suggests:

- an underscore is used to separate words that would normally have a space in natural language;
- the schema prefix of the attribute is retained, presented in lower case and separated by an underscore, and
- camel case is converted into lower case, and again using underscores to separate words.

To move from OIDC to SAML, the reverse is applied. The following table shows the example given in the white paper:

SAML attribute	**OIDC claim**
eduPersonFooBar	eduperson_foo_bar
SchacFooBar	schac_foo_bar
voPersonFooBar	voperson_foo_bar

Research and Education WG

Other

Patterns / Best practices

This section outlines a the advice and best practices that can be given at the time of writing this document.

Obtaining a token

Depending on the use case, different tokens may be desired and different flows may be suitable to obtain the desired token.

Web flows are typically based on the Authorisation Code Flow. In this flow, the user points a web browser to the desired service (“client” in OIDC specification). The client redirects the user to the OP for authentication. The OP redirects the user back to the client together with the Authorisation Code, which enables the service to obtain several tokens from the authorization endpoint of the OP. The tokens returned depend on what was agreed upon between client and

OP at the time of registering the client with the OP. The same holds for the scopes made available to the client (they must be a subset of the scopes agreed during the registration). The user is informed about the information and which scopes will be released from the OP to the client.

The set of returned tokens includes the ID Token and the Access Token, and in case the “offline scope” was requested also the Refresh Token.

Other use cases, such as Commandline or API Calls may require the use of different flows, or the use of an additional component, which is able to run a web flow and to store the Refresh Token for subsequent obtaining of the Access Tokens from the OP.

Verifying a token

Depending on the token, different measures need to be taken.
The ID Token is a JWT[RFC7519], i.e. a signed JSON document that can be verified offline, using available public key information.

The Access Token and the Refresh Token are opaque strings that need to be verified at the issuing OP. For the Refresh Token, the token endpoint of the OP is used. If the token is Valid, a valid response will be returned, which includes an Access Token (and optionally a new Refresh Token). Note, that the Refresh Token is bound to a specific client_id, i.e. only the given client_id can obtain and use the refresh token and, furthermore, requires a client_secret.

Verification of an Access Token:
For this, multiple possibilities exist:

- In case the Access Token is a JWT, offline verification is possible, and sufficient.
- Otherwise the access token can be verified via token endpoint or the token introspection endpoint => Client_id / secret required may be used
- In case no client_id/client_secret are available, accessing the userinfo endpoint at the OP is an option. This only works with a valid Access Token. In addition the userinfo (i.e. the scopes available for the given Access Token) will be returned by the OP.

In case of services supporting multiple OPs JWTs...

Caching

According to the specification tokens are exclusively verified by the issuer. This however poses a bottleneck, specifically in large deployments. Most applications will

ID Token: The ID token contains information about the authentication. E.g. assurance, claims requested (or obtained??).
AccessToken: The AT per se does not contain any information unless it is a JWT. The AT can be used to access the userinfo endpoint at the OP. If it's a JWT...

- You SHOULD cache information requested from the OP for a reasonable time to avoid overloading it.
- You do (do not??) register a client_id/client_secret for every instance that needs to verify an AT.
- You NEED TO register cid/cs for every instance that needs to get an ID-Token or to do token introspection
- ...
- RFC7523 => JSON Web Token Profile for OAuth2

Summary

References

[OIDC-CORE]	OpenID Connect Core 1.0 https://openid.net/specs/openid-connect-core-1_0.html
[OIDCRE-WHITE]	White Paper for implementation of mappings between SAML 2.0 and OpenID Connect in Research and Education https://wiki.refeds.org/display/CON/Consultation%3A+SAML2+and+OIDC+Mappings
[OIDF]	OpenID Connect Federation 1.0 - draft 07 https://openid.net/specs/openid-connect-federation-1_0.html
[RANDE]	Research & Education (R&E) WG https://openid.net/wg/rande
[RFC2119]	Key words for use in RFCs to indicate Requirement levels https://tools.ietf.org/html/rfc2119
[RFC6749]	The OAuth 2.0 Authorisation Framework https://tools.ietf.org/html/rfc6749
[RFC7519]	JSON Web Token (JWT) https://tools.ietf.org/html/rfc7519
[RFC7662]	OAuth 2.0 Token Introspection https://tools.ietf.org/html/rfc7662
[TOKEN-EXCHG]	Draft RFC on OAuth2.0 Token Exchange https://datatracker.ietf.org/doc/draft-ietf-oauth-token-exchange

Old

Recap of previous discussions

- ○ Currently there is no unified way to express user profile information (in all details, in non-SAML use-cases). Therefore, we here provide a list of references of examples of conveying user profile information from different communities
    - Updates on WLCG Authorisation: Description of the new system / scitokens
    - Human genome data community has a similar profile
    - Macaroons
- We should probably also give a recap o the flow that we envisage to be fulfilled.
- Doc should be as concise as G027. (Think AEGIS discussion)

Preparation Material

- Rande
    - https://openid.net/wg/rande/
- OIDCRE: => Consultation closed.
    - https://wiki.refeds.org/display/CON/Consultation%3A+SAML2+and+OIDC+Mappings
    - White Paper Google Doc
- RAF: => Should not be in the way
    - https://wiki.refeds.org/display/ASS/REFEDS+Assurance+Framework+ver+1.0
- JRA1.2 b: (From 2017) https://docs.google.com/document/d/1EURj8VsTaAOodSOIzfLZ08v_DHWoKqmTSe0O8fpy35g

TODOS for this doc (from Call on March 6)

- Distinguish cases:
    - Getting the token(s)
    - Verifying the token(s)
    - Which information is in which token?
    - Which information to use (oidcre and stuff)

Appendix

Appendix A: OpenID Connect Flows:

OpenID Connect (OIDC) defines several flows. Those relevant for this document are listed here. The OIDC specification is extensible, so that we cannot provide an exhaustive list references here.

Authorisation Code flow

The authorisation code flow (or grant) is defined by RFC6749 and is an OAuth2 flow. As such, it is also used for OIDC use cases, and it is also defined in OIDC specification.

In short, it uses an authorisation server as an intermediary between the client and resource owner (or “End user” and “RP”). The benefits are that the RP credentials are never shared with the End User, and End user credentials are never shared with the RP (in addition to all the other SSO benefits).

There are several parameters that must or can be used with this flow, and they can all be found in the specification (OIDC and/or OAuth2). The MUST ones are required, and therefore must always be defined. We also follow the spec in recommending the usage of “state” parameter, and furthermore recommend the following parameters:

- max_age:
- etc...

Device Code

The Device Code grant type is used by browserless or input-constrained devices in the device flow to exchange a previously obtained device code for an access token.[11]
The Device Code grant type value is urn:ietf:params:oauth:grant-type:device_code,[12] i.e. used as a parameter “grant_type=urn:ietf:params:oauth:grant-type:device_code”. In this case, the authorisation can be performed on a secondary device, that has the requisite input and browser capabilities for an OAuth flow. This device flow can also be used for CLI use cases.

The necessary parameters and potential security implications can be found in the spec. Here, we would like to emphasise certain points:

[11] https://oauth.net/2/grant-types/device-code/
[12] https://tools.ietf.org/html/draft-ietf-oauth-device-flow-14

- Verification Code should be short, but generated in a non-guessable way. Limiting the guessing rate is also a good practice. It is also permitted to use QR codes or similar, but still the code should also be displayed as text. Also, the lifetime should be reasonably long, but not too long, in order to reduce the phishing attacks.
- Verification URI should be easy to input.
- To reduce the likelihood of phishing, the user should be informed that they are authorising a device during the flow, and to be sure that they do control the device in question.
- Follow best security practices, i.e. protect the secrets, and again we recommend using the “state” parameter

Spec also include other recommendations, and the authors feel they should also be followed.

Implicit flow

The Implicit grant type is a simplified flow that can be used by public clients, where the access token is returned immediately without an extra authorisation code exchange step.[13]
It is generally not recommended to use the implicit flow, better practice is to use public clients with PKCE instead.
Furthermore, hybrid and password flow are also not recommended.

Token Exchange

Token Exchange, or full name “OAuth 2.0 Token Exchange”[14] is a specification on how to request and obtain security tokens, including tokens employing impersonation and delegation. While still in draft phase, it is nevertheless an important feature to support. The full specification will not be repeated here, however some important points will be summarised.

There are several scenarios when token exchange is necessary, such as for situations when just passing a token downstream is not possible or not recommended, as in situation when the token is scoped to only one instance, or when there is a need to “act” on behalf of the user for long running jobs, or when there is a need to obtain e.g. a refresh token. Token exchange provides a protocol for these situations.

One important point of distinction is “impersonation” vs “delegation”. In the “impersonation” scenario, when subject A impersonates B, A has all the rights of B and is indistinguishable from B. When A interacts with any entity, A is B. In the “delegation” scenario, A still has its own identity, distinguishable from B. Now when A interacts with some other entity, it is unambiguous that A is representing B, and that B has granted some (but not necessarily all) rights to A, as explained in the spec.

[13] https://oauth.net/2/grant-types/implicit/
[14] https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-16

In order to request a token exchange, a client must support "urn:ietf:params:oauth:grant-type:token-exchange" grant type. Furthermore, OPs that want to support this protocol, must provide a token endpoint (typically "/token"). The request must be in the "application/x-www-form-urlencoded" format. There are several parameters to be included in the request, of which "grant_type", "subject_token", and "subject_token_type" are REQUIRED, and others optional. However, it is always a good practice to include more information, e.g. parameter "resource" should be used if the resource is known, parameter "scope" should be used to request only the necessary scopes, etc.

For the response, "access_token", "issued_token_type", and "token_type" are REQUIRED parameters. We also recommend (as in the spec) to include "expires_in", and we recommend to include "scope". Parameter "refresh_token" should be issued only if requested. We also recommend to issue tokens as JWT[15].

Proof Key for Code Exchange by OAuth Public Clients (PKCE)

OAuth 2.0 public clients utilizing the Authorisation Code Grant are susceptible to the authorisation code interception attack. This specification describes the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy").[16] This is possible since for public clients the client secret is not confidential. Furthermore, absence of TLS (and certificate pinning) in the communication path is also enabling this attack.

```
                                                 +-------------------+
                                                 |   Authz Server    |
       +--------+                                | +---------------+ |
       |        |--(A)- Authorization Request ---->|               | |
       |        |       + t(code_verifier), t_m  | | Authorization | |
       |        |                                | |    Endpoint   | |
       |        |<-(B)---- Authorization Code -----|               | |
       |        |                                | +---------------+ |
       | Client |                                |                   |
       |        |                                | +---------------+ |
       |        |--(C)-- Access Token Request ---->|               | |
       |        |          + code_verifier       | |    Token      | |
       |        |                                | |   Endpoint    | |
       |        |<-(D)------ Access Token ---------|               | |
       +--------+                                | +---------------+ |
                                                 +-------------------+
```

PKCE Protocol Flow

[15] https://tools.ietf.org/html/rfc7519
[16] https://tools.ietf.org/html/rfc7636

A. The client creates and records a secret named the "code_verifier" and derives a transformed version "t(code_verifier)" (referred to as the "code_challenge"), which is sent in the OAuth 2.0 Authorisation Request along with the transformation method "t_m".

B. The Authorisation Endpoint responds as usual but records "t(code_verifier)" and the transformation method.

C. The client then sends the authorisation code in the Access TokenRequest as usual but includes the "code_verifier" secret generated at (A).

D. The authorisation server transforms "code_verifier" and compares it to "t(code_verifier)" from (B). Access is denied if they are not equal.

An attacker who intercepts the authorisation code at (B) is unable to redeem it for an access token, as they are not in possession of the "code_verifier" secret.

Client Credentials

In some cases, applications may need an access token to act on behalf of themselves rather than a user. In this case, applications need a way to get an access token for their own account, outside the context of any specific user. OAuth provides the “client_credentials” grant type for this purpose. More information can be found in the spec, section 4.4.