Basics of OAUTH

<https://tools.ietf.org/html/rfc6749>

# **Roles in OAUTH 2.0 : OAuth defines four roles:**

**resource owner**

An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user. The resource owner is the person or application that owns the data that is to be shared. For example, a user on Facebook could be a resource owner. The resource they own is their data. The resource owner is typically a person, but could also be an application.

**resource server**

The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens. The resource server is the server hosting the resource owner's data. For example, Facebook is a resource server.

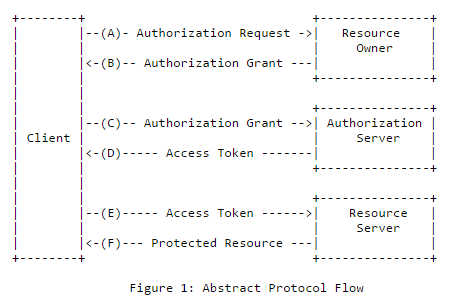
**client**

An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices). The client application is the application requesting access to the resources stored on the resource server. The resources, which are owned by the resource owner. A client application could be a game requesting access to a user's Facebook account. Example lik Flipkart or Bigbasket, makemytrip

**authorization server**

The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. Facebook can be authorization server.

# **Protocol Flow**



The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the four roles and includes the following steps:

1. The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.
2. The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.
3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
6. The resource server validates the access token, and if valid, serves the request.

**Authorization Grant**

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token.

This specification defines four grant types --

1. **authorization code,**
2. **implicit,**
3. **resource owner password**
4. **credentials**
5. **Authorization code** (aka: Web Application flow, aka: Server-Side Web Application Flow)  
   This grant type most often associated with OAuth. Client application must redirect user to the authorization server to authorize access to their data. After user has approved access, it’s get redirected back to client application with authorization code as a parameter. The code then is exchanged for access token with a separate call from client application to Authorization server token endpoint. After access token is issued, it can be used by client application to access protected resources on Resource Server.  
   *Best fits for:* Web apps or native mobile apps with browser support  
   *Type:* Redirect
6. **Implicit grant** (aka: Client-Side Web Applications Flow)  
   Simplified version of Authorization code grand type. Here instead of issuing the client an authorization code, the client is issued an access token directly.  
   *Best fits for:* Browser-based apps running entirely in the browser after loading the source code from a web page  
   *Type:* Redirect
7. **Resource Owner grant** (aka: Password-based grant)  
   User (resource owner) has to give its credentials directly to the client application.  
   *Best fits for:* Highly trusted client apps, such as mobile apps written by API provider.  
   *Type:* Password
8. **Client Credentials grant**Used when the client app is requesting access to protected resources, such as storage service or database service, on behalf of ***itself*** rather than on behalf of a specific user.  
   *Type:* Client App Credentials

**Access Token**

Access tokens are credentials used to access protected resources.

**Refresh Token**

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.

# **Error Response**

The authorization server responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

**invalid\_request** : The request is missing a required parameter

**invalid\_client :** Client authentication failed (e.g., unknown client, no client authentication included, or unsupported

authentication method). The authorization server MAY return an HTTP 401 (Unauthorized) status code to indicate

which HTTP authentication schemes are supported.

**invalid\_grant** : The provided authorization grant (e.g., authorization code, resource owner credentials) or refresh token is invalid, expired, revoked, does not match the redirection URI used in the authorization request, or was issued to

another client.

**unauthorized\_client** : The authenticated client is not authorized to use this authorization grant type.

**unsupported\_grant\_type** : The authorization grant type is not supported by the authorization server.

**Invalid\_scope :** The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

2 Legged VS 3 Legged

**Grant Types used in 2-Legged and 3-Legged**

**2-Legged**

The following grant types are used in the 2-Legged OAuth scenario.

| **Grant Type** | **Description** |
| --- | --- |
| Client Credentials | The client presents its own credentials to the OAuth Authorization Server in order to obtain an access token. This access token is either associated with the client’s own resources, and not a particular resource owner, or is associated with a resource owner for whom the client is otherwise authorized to act. |
| Resource Owner Password Credentials | The client collects the resource owner’s password and exchanges it at the OAuth AS for an access token, and often a refresh token (see below). This grant type is suitable in cases where the RO has a trust relationship with the client, such as its computer operation system or a highly privileged application since the client must discard the password after using it to obtain the access token. |

**3-Legged**

The following grant types are used in the 3-Legged OAuth scenario.

| **Grant Type** | **Description** |
| --- | --- |
| Authorization Code | An authorization code is returned to the client through a browser redirect after the resource owner gives consent to the OAuth Authorization Server. The client then exchanges the authorization code for an access token. Resource owner credentials are never exposed to the client. |
| Implicit | An access token is returned to the client through a browser redirect in response to the resource owner authorization request.  This grant type is suitable for clients that do not support keeping client credentials confidential (for use in authenticating with the OAuth Authentication Server) such as client applications implemented in a browser using a scripting language like JavaScript. |

**Example of OAUTH 2.0 2-Legged Endpoints**

**GET**

[**http://localhost:8080/bankapp-spring-jersey-oauth2/oauth/token**](http://localhost:8080/bankapp-spring-jersey-oauth2/oauth/token)**?**

**grant\_type=password&client\_id=restapp&client\_secret=restapp&username=piku&password=piku**

**Response**

**{"access\_token":"16ce6fc2-f3e1-4842-a2d9-79b5bd26de11","token\_type":"bearer","refresh\_token":"eb1d2526-f54f-444b-8d3f-b515de5627c0","expires\_in":119}**

**Example of OAUTH 2.0 3-Legged Endpoints**

[**https://oauth\_service/login/oauth/authorize**](https://oauth_service/login/oauth/authorize)**?**

**client\_id=abcd123&redirect\_uri=http://localhost/oauth/code\_callback&scope=user**

**Actually It happens like this**

[**http://localhost:8080/zara-auth/oauth/authorize**](http://localhost:8080/zara-auth/oauth/authorize)**?**

**response\_type=token&client\_id=client1&scope=read%20write%20delete&redirect\_uri=http://localhost:8080/zara-client/index.jsp**

[**http://localhost:8080/zara-auth/login.do**](http://localhost:8080/zara-auth/login.do)

[**http://localhost:8080/zara-auth/oauth/authorize**](http://localhost:8080/zara-auth/oauth/authorize)

[**http://localhost:8080/zara-client/index.jsp#**](http://localhost:8080/zara-client/index.jsp)

**access\_token=9cef85e1-ee81-44b1-87ce-0bfdf836cfc3&token\_type=bearer&expires\_in=3555**

**How 2-Legged and 3-Legged works**

**3-Legged:**

In 3-Legged OAuth a resource owner wants to give a client access to a server without sharing their credentials (i.e. username/password). For example, a user (resource owner) wants to give a third-party application (client) access to his Twitter account (server).

This scenario works as follows:

* Client signs up to the server and obtains client credentials (i.e., consumer key and secret).
* User wants to give the client access to his protected resources on the server.
* Client retrieves the temporary credentials (i.e., request token) from the server.
* Client redirects the resource owner to the server.
* Resource owner grants the client access to his protected resources on the server.
* Server redirects the user back to the client.
* Client uses the temporary credentials to retrieve the token credentials (i.e., access token) from the server.
* Client uses the token credentials to access the protected resources on the server.

**2-Legged:**

2-Legged OAuth describes a typical client-server scenario without any user involvement. For example, a local Twitter client application accessing your Twitter account.

This 2-legged OAuth scenario consists of the first and last steps of 3-legged OAuth:

* Client has signed up to the server and got his client credentials (i.e., consumer key and secret).
* Client uses his client credentials (and empty token credentials) to access the protected resources on the server.

### What is an Access Token?

An access token is issued as a string that represents an access authorization issued to the client. It is used by the client to access protected resources hosted by the resource server. Access tokens are issued to clients by an authorization server with the approval of the resource owner. Each access token includes an expiration attribute that indicates how long the token is valid.

Note: An access token timeout is typically 15 days (or 1296000 seconds).

Three legged does not imply a certain type of app as in "browser based". Three legged means that an application acts on the direct behalf of a user. In the three legged scenarios there is

1. an application (consumer),
2. a user (resource owner) and
3. an API (service provider).

In two legged scenarios there is no concept of a user. Typically this has to do with application-to-application solutions. There the application (consumer) acts on behalf of itself. So in two legged OAuth, there is:

1. an application (consumer),
2. an API (service provider)

The difference is simply that there is no need of a user authorisation step in the 2-legged approach.

**OAuth 2.0:**

There are two types of access tokens:

* **Bearer Token**—This access token type is a security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession).
* **Message Authentication Code (Mac)**—This access token type is a security code that is typed in by the user of a computer to access accounts or portals. This code is attached to the message or request sent by the user. Message Authentication Codes (MACs) attached to the message must be recognized by the receiving system in order to grant the user access. MACs are commonly used in electronic funds transfers (EFTs) to maintain information integrity.

**Token End Points**

**OAuth 2.0 Endpoints**:

* **Authorization Endpoint** is the endpoint on the authorization server where the resource owner provides credentials, such as username and password, in and grants authorization to the client app to access the resources or a specified subset of the resources.
* **Token Endpoint** is the endpoint on the authorization server where the client application exchanges the authorization code, client ID, and client secret and receives in exchange an access token which allows the app to access the approved resources.
* **Redirection Endpoint** is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint.

OAuth 1.0a supports the following endpoints:

* **Request Token Endpoint URL**—A Request Token is used by the consumer to ask the user to authorize access to the protected resources. The Request Token Endpoint URL is used to obtain an unauthorized Request Token.
* **User Authorization URL**—The consumer cannot use the Request Token until it has been authorized by the user. The User Authorization URL is used to obtain user authorization for Consumer access.
* **Access Token Endpoint URL**—An Access Token is a value used by the consumer to gain access to the protected resources on behalf of the user, instead of using the user's service provider credentials. The Access Token Endpoint URL is used to exchange the user-authorized Request Token for an Access Token.

**Difference between OAUTH 1.0 and OAUTH 2.0**

<http://stackoverflow.com/questions/4113934/how-is-oauth-2-different-from-oauth-1>

**More OAuth Flows to allow better support for non-browser based applications.** This is a main criticism against OAuth from client applications that were not browser based. For example, in OAuth 1.0, desktop applications or mobile phone applications had to direct the user to open their browser to the desired service, authenticate with the service, and copy the token from the service back to the application. The main criticism here is against the user experience. With OAuth 2.0, there are now new ways for an application to get authorization for a user.

**OAuth 2.0 no longer requires client applications to have cryptography.** This hearkens back to the old Twitter Auth API, which didn't require the application to HMAC hash tokens and request strings. With OAuth 2.0, the application can make a request using only the issued token over HTTPS.

**OAuth 2.0 signatures are much less complicated.** No more special parsing, sorting, or encoding.

**OAuth 2.0 Access tokens are "short-lived".** Typically, OAuth 1.0 Access tokens could be stored for a year or more (Twitter never let them expire). OAuth 2.0 has the notion of refresh tokens. While I'm not entirely sure what these are, my guess is that your access tokens can be short lived (i.e. session based) while your refresh tokens can be "life time". You'd use a refresh token to acquire a new access token rather than have the user re-authorize your application.

**Finally, OAuth 2.0 is meant to have a clean separation of roles between the server responsible for handling OAuth requests and the server handling user authorization.**

OAuth 2.0 signatures are not required for the actual API calls once the token has been generated. It has only one security token.

OAuth 1.0 requires client to send two security tokens for each API call, and use both to generate the signature. It requires the protected resources endpoints have access to the client credentials in order to validate the request.

**Difference between Resource Server and Authorization Server**

The reason this is out of scope for the specification is the wide range of ways to accomplish this connection between the two entities. The main question is how complex is your deployment.

For example, do you have one server managing authentication and access, and a set of discrete services each with its own servers serving the API calls? Or, do you have just one box with one web server which handles both authentication/authorization and the API calls?

In the case of a single box, not much is needed as the entity issuing tokens is the same as the one validating them. You can implement tokens to use a database table key and lookup the record in the database (or memory cache) on every request, or you can encode the scope, user id and other information directly into the token and encrypt it using a symmetric or asymmetric algorithm.

Things get a bit more complex when dealing with a distributed environment, but not by much. You still issue tokens at the authorization server, but the resource server needs a way to validate those. It can do it by making an internal API available to the resource server to ask the authorization server to "resolve" the token (which can be fast in a local environment), or the two can establish a public/private key pair or symmetric secret and use that to encrypt everything the resource server needs into the token.

Self contained tokens are longer but offer much better performance per-request. However, they come with a price - you can't really revoke them while they are still valid (not expired). For this reason, self contained tokens should be very short lived (whatever is acceptable to you to leave access open after it was revoked - e.g. many sites use one hour), with a refresh token good for a year or more to get new tokens.

**In OAuth 2.0, how do resource servers assert a token issued by an authorization server?**

The OAuth 2.0 spec, RFC6749, very specifically punts on this issue in section 7:

The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification but generally involve an interaction or coordination between the resource server and the authorization server.

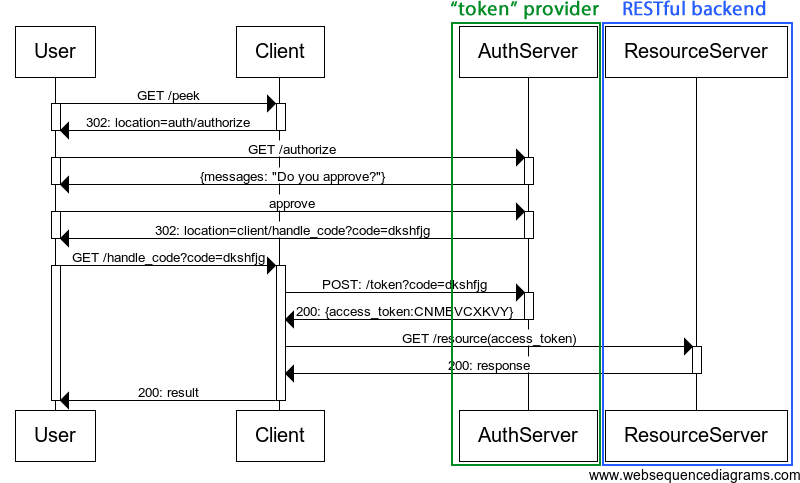
<http://stackoverflow.com/questions/9891469/oauth2-token-validation-verification-in-spring>

The Resource Server's OAuth2ProtectedResourceFilter should validate the token against the same database that the Authorization Server used to store the token.

By Dave Syer

I think most people doing this are using JdbcTokenStore (or a custom TokenStore) shared between the 2 servers. A remote token service would be quite practical I think, but you need another endpoint which isn't part of the spec, so we haven't added it to Spring Security OAuth as yet. If you want a remote token service you could check out this (and the associated endpoint on the auth server):

OAUTH Interaction Diagram



# References

<http://stackoverflow.com/questions/13369516/why-is-there-3-legged-oauth2-when-2-legged-works-so-well>

<http://docs.akana.com/cm/learnmore/api_admin_oauth_support.htm#what_are_the_oauth_20_endpoints_and_how_do_they_work>

**What is OAUTH ?**

<https://en.wikipedia.org/wiki/OAuth>

**OAuth** is an [open standard](https://en.wikipedia.org/wiki/Open_standard) for [authorization](https://en.wikipedia.org/wiki/Authorization). OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP), OAuth essentially allows[access tokens](https://en.wikipedia.org/wiki/Access_token) to be issued to third-party clients by an authorization server, with the approval of the resource owner. The client then uses the access token to access the protected resources hosted by the resource server.[[1]](https://en.wikipedia.org/wiki/OAuth#cite_note-1) OAuth is commonly used as a way for Internet users to log into third party websites using their Microsoft, Google, Facebook or Twitter accounts without exposing their password.[[2]](https://en.wikipedia.org/wiki/OAuth#cite_note-2)

OAuth is a service that is complementary to and distinct from [OpenID](https://en.wikipedia.org/wiki/OpenID). OAuth is also distinct from [OATH](https://en.wikipedia.org/wiki/Initiative_For_Open_Authentication), which is a *reference architecture* for *authentication*, not a *standard* for *authorization*. However, OAuth is directly related to [OpenID Connect (OIDC)](https://en.wikipedia.org/wiki/OpenID_Connect)since OIDC is an authentication layer built on top of OAuth 2.0.

OAuth 2.0 is the next evolution of the OAuth protocol and is not backwards compatible with OAuth 1.0. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. The specification and associated RFCs are developed by the IETF OAuth WG;[[5]](https://en.wikipedia.org/wiki/OAuth#cite_note-5) the main framework was published in October 2012. (It was expected to be finalized by the end of 2010, according to [Eran Hammer](https://en.wikipedia.org/w/index.php?title=Eran_Hammer&action=edit&redlink=1).[[6]](https://en.wikipedia.org/wiki/OAuth#cite_note-6) However, due to discordant views about the evolution of OAuth, Hammer left the working group.[[7]](https://en.wikipedia.org/wiki/OAuth#cite_note-oauthtwoandtheroadtohell-7))

[Facebook](https://en.wikipedia.org/wiki/Facebook)'s [Graph API](https://en.wikipedia.org/wiki/Facebook_Platform#Graph_API) only supports OAuth 2.0.[[8]](https://en.wikipedia.org/wiki/OAuth#cite_note-8) [Google](https://en.wikipedia.org/wiki/Google) supports OAuth 2.0 as the recommended authentication mechanism for all of its APIs.[[9]](https://en.wikipedia.org/wiki/OAuth#cite_note-9) As of 2011[Microsoft](https://en.wikipedia.org/wiki/Microsoft)[[10]](https://en.wikipedia.org/wiki/OAuth#cite_note-10) has added OAuth 2.0 experimental support to their APIs.

The OAuth 2.0 Framework[[11]](https://en.wikipedia.org/wiki/OAuth#cite_note-Dick_Hardt-11) and Bearer Token Usage[[12]](https://en.wikipedia.org/wiki/OAuth#cite_note-12) were published in October 2012. Other documents are still being worked on within the OAuth working group.

<http://stackoverflow.com/questions/4201431/what-exactly-is-oauth-open-authorization>

OAuth allows notifying a **resource provider** (e.g. Facebook) that the **resource owner** (e.g. you) grants permission to a **third-party** (e.g. a Facebook Application) access to their **information** (e.g. the list of your friends).

If you read it stated as plainly, I would understand your confusion. So let's go with a concrete example: joining yet another social network!

Say you have an existing GMail account. You decide to join LinkedIn. Adding all of your *many, many*friends manually is tiresome and error-prone. You might get fed up half-way or insert typos in their e-mail address for invitation. So you might be tempted not to create an account after all.

Facing this situation, LinkedIn has the Good Idea(TM) to write a program that adds your list of friends automatically because computers are far more efficient and effective at tiresome and error prone tasks. Since joining the network is now so easy, there is *no way* you would refuse such an offer, now would you?

Without an API for exchanging this list of contacts, you would have to give LinkedIn the username and password to your GMail account, thereby **giving them too much power**.

This is where OAuth comes in. If your GMail supports the OAuth protocol, then LinkedIn can ask you to authorize them to access your GMail list of contacts.

OAuth allows for:

1. Different access levels: read-only VS read-write. This allows you to grant access to your user list or a bi-directional access to automatically synchronize your new LinkedIn friends to your GMail contacts.
2. Access granularity: you can decide to grant access to only your contact information (uername, e-mail, date of birth, etc.) or to your entire list of friends, calendar and what not.
3. It allows you manage access from the resource provider's application. If the third-party application does not provide mechanism for cancelling access, you would be stuck with them having access to your information. With OAuth, there is provision for revoking access at any time.

Will it become a de facto (standard?) in near future?

Well, although OAuth is a significant step forward, it doesn't solve problems if people don't use it correctly. For instance, if a resource provider gives only a single read-write access level to all your resources at once and doesn't provide mechanism for managing acces, then there is no point to it.

In practice, it fits the social network model very well. It is especially popular for those social networks that want to allow third-party "plugins". This is an area where access to the resources is inherently necessary and is also inherently unreliable (i.e. you have little or no quality control over those applications).

I haven't seen so many other uses out in the wild. I mean, I don't know of an online financial advice firm that will access your bank records automatically, although it *could* technically be used that way.

Some OAuth Concepts

<https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified>

**Roles**

**The Third-Party Application: "Client"**

The client is the application that is attempting to get access to the user's account. It needs to get permission from the user before it can do so.

**The API: "Resource Server"**

The resource server is the API server used to access the user's information.

**The User: "Resource Owner"**

The resource owner is the person who is giving access to some portion of their account.

## Creating an App

Before you can begin the OAuth process, you must first register a new app with the service. When registering a new app, you usually register basic information such as application name, website, a logo, etc. In addition, you must register a redirect URI to be used for redirecting users to for web server, browser-based, or mobile apps.

### Redirect URIs

The service will only redirect users to a registered URI, which helps prevent some attacks. Any HTTP redirect URIs must be protected with TLS security, so the service will only redirect to URIs beginning with "https". This prevents tokens from being intercepted during the authorization process.

### Client ID and Secret

After registering your app, you will receive a client ID and a client secret. The client ID is considered public information, and is used to build login URLs, or included in Javascript source code on a page. The client secret **must** be kept confidential. If a deployed app cannot keep the secret confidential, such as Javascript or native apps, then the secret is not used.

## Authorization

The first step of OAuth 2 is to get authorization from the user. For browser-based or mobile apps, this is usually accomplished by displaying an interface provided by the service to the user.

OAuth 2 provides several "grant types" for different use cases. The grant types defined are:

* **Authorization Code** for apps running on a [web server](https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified#web-server-apps)
* **Implicit** for [browser-based](https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified#browser-based-apps) or [mobile apps](https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified#mobile-apps)
* **Password** for logging in with a [username and password](https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified#other-app-types)
* **Client credentials** for [application access](https://aaronparecki.com/articles/2012/07/29/1/oauth2-simplified#other-app-types)

### Password

OAuth 2 also provides a "password" grant type which can be used to exchange a username and password for an access token directly. Since this obviously requires the application to collect the user's password, it should only be used by apps created by the service itself. For example, the native Twitter app could use this grant type to log in on mobile or desktop apps.

To use the password grant type, simply make a POST request like the following:

POST https://api.oauth2server.com/token

grant\_type=password&

username=USERNAME&

password=PASSWORD&

client\_id=CLIENT\_ID

### Application access

In some cases, applications may wish to update their own information such as their website URL or application icon, or they may wish to get statistics about the users of the app. 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.

To use the client credentials grant type, make a POST request like the following:

POST https://api.oauth2server.com/token

grant\_type=client\_credentials&

client\_id=CLIENT\_ID&

client\_secret=CLIENT\_SECRET

You will get an access token response in the same format as the other grant types

<http://www.bubblecode.net/en/2013/03/10/understanding-oauth2/>

<https://developer.salesforce.com/page/Digging_Deeper_into_OAuth_2.0_on_Force.com>

<https://apigility.org/documentation/auth/authentication-oauth2>

<https://msdn.microsoft.com/en-in/library/azure/dn645542.aspx>

<http://www.3pillarglobal.com/insights/api-management-securing-apis>