**OAuth 2.0 Tutorial**

<http://tutorials.jenkov.com/oauth2/index.html>

OAuth 2.0 is an open authentication protocol which enables applications to access each other’s data. For instance, a game application can access a user’s data in the Facebook application, or a location based application can access the user data of the Foursquare application etc.

OAuth 2.0 enables a user to login to a single application (e.g. Google, Facebook Foursquare, Twitter etc.), and share their data in that application with other applications.

Here is a diagram illustrating the principle:

|  |
| --- |
| Example of how OAuth 2.0 is used to share data via applications. |
| **Example of how OAuth 2.0 is used to share data via applications.** |

The user accesses the game web application. The game web application asks the user to login to the game via Facebook. The user logs into Facebook, and is sent back to the game. The game can now access the users data in Facebook, and call functions in Facebook on behalf of the user (e.g. posting status updates).

**OAuth 2.0 Use Cases**

OAuth 2.0 can be used either to create an application that can read user data from another application (e.g. the game in the diagram above), or an application that enables other applications to access its user data (e.g. Facebook in the example above).

OAuth 2.0 is a replacement for OAauth 1.0, which was more complicated. OAuth 1.0 involved certificates etc. OAuth 2.0 is simpler. It requires no certificates at all, just SSL / TLS.

**OAuth 2.0 Specification**

The purpose of this tutorial is to provide an overview of the OAuth 2.0 protocol that is easy to understand. It is not the purpose to describe every detail of the specification, though.

If you plan to implement OAuth 2.0 you will most likely need to visit the specification to study it in full detail. You can find the specification here:

[**http://tools.ietf.org/html/draft-ietf-oauth-v2-23**](http://tools.ietf.org/html/draft-ietf-oauth-v2-23)

**Table of Contents**

Here is a list of the topics covered in this OAuth 2.0 trail. This list (menu) is also present at the top right of every page in the trail.

|  |  |  |
| --- | --- | --- |
| **OAuth 2.0 Tutorial** | | |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Introduction**](http://tutorials.jenkov.com/oauth2/index.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Overview**](http://tutorials.jenkov.com/oauth2/overview.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Roles**](http://tutorials.jenkov.com/oauth2/roles.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Client Types**](http://tutorials.jenkov.com/oauth2/client-types.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Authorization**](http://tutorials.jenkov.com/oauth2/authorization.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Endpoints**](http://tutorials.jenkov.com/oauth2/endpoints.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [**OAuth 2.0 Requests and Responses**](http://tutorials.jenkov.com/oauth2/request-response.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [**Authorization Code Grant**](http://tutorials.jenkov.com/oauth2/authorization-code-request-response.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [**Implicit Grant**](http://tutorials.jenkov.com/oauth2/implicit-request-response.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [**Resource Owner Credentials**](http://tutorials.jenkov.com/oauth2/resource-owner-credentials-request-response.html) |
|  |  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png [**Client Credentials**](http://tutorials.jenkov.com/oauth2/client-credentials-request-response.html) |

# OAuth 2.0 Overview

As mentioned in the introduction, OAuth 2.0 is an open authentication protocol which enables applications to access each other’s data. Here I will try to provide an overview of how the protocol works, and the various concepts mentioned in the specification.

OAuth 2.0 covers different ways a client application can obtain authorization to access the resources stored on the resource server. Here I will show you the most common and most secure use case: A client web application requesting access to resources in another web application.

This diagram shows the authentication process:

|  |
| --- |
| Example of how OAuth 2.0 is used to share data via applications. |
| **Example of how OAuth 2.0 is used to share data via applications.** |

First the user accesses the client web application. In this web app is button saying "Login via Facebook" (or some other system like Google or Twitter).

Second, when the user clicks the login button, the user is redirected to the authenticating application (e.g. Facebook). The user then logs into the authenticating application, and is asked if she wants to grant access to her data in the authenticating application, to the client application. The user accepts.

Third, the authenticating application redirects the user to a redirect URI, which the client app has provided to the authenticating app. Providing this redirect URI is normally done by registering the client application with the authenticating application. During this registration the owner of the client application registers the redirect URI. It is also during this registration that the authenticating application gives the client application a client id and a client password.

Fourth, the user accesses the page located at the redirect URI in the client application. In the background the client application contacts the authenticating application and sends the client id and password which was given to the client application along with the authentication code. The authentication application responds with an access token.

Once the client application has obtained an access token, this access token can be sent to the Facebook, Google, Twitter etc. to access resources in these systems, related to the user who logged in.

# OAuth 2.0 Roles

OAuth 2.0 defines the following roles of users and applications:

* Resource Owner
* Resource Server
* Client Application
* Authorization Server

These roles are illustrated in this diagram:

|  |
| --- |
| OAuth 2.0 roles as defined in the specification. |
| **OAuth 2.0 roles as defined in the specification.** |

The resource owner is the person or application that owns the data that is to be shared. For instance, a user on Facebook or Google could be a resource owner. The resource they own is their data. The resource owner is depicted in the diagram as a person, which is probably the most common situation. The resource owner could also be an application. The OAuth 2.0 specification mentions both possibilities.

The resource server is the server hosting the resource owned by the resource owner. For instance, Facebook or Google is a resource server (or has a resource server).

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.

The authorization server is the server authorizing the client app to access the resources of the resource owner. The authorization server and the resource server can be the same server, but it doesn't have to. The OAuth 2.0 specification does not say anything about how these two servers should communicate, if they are separate. This is an internal design decision to be made by the resource server + authorization server developers.

# OAuth 2.0 Client Types

The OAuth 2.0 client role is subdivided into a set of client types and profiles. This text will explain these types and profiles.

The OAuth 2.0 specification defines two types of clients:

* Confidential
* Public

A confidential client is an application that is capable of keeping a client password confidential to the world. This client password is assigned to the client app by the authorization server. This password is used to identify the client to the authorization server, to avoid fraud. An example of a confidential client could be a web app, where no one but the administrator can get access to the server, and see the client password.

A public client is an application that is not capable of keeping a client password confidential. For instance, a mobile phone application or a desktop application that has the client password embedded inside it. Such an application could get cracked, and this could reveal the password. The same is true for a JavaScript application running in the user’s browser. The user could use a JavaScript debugger to look into the application, and see the client password.

## Client Profiles

The OAuth 2.0 specification also mentions a set of client profiles. These profiles are concrete types of applications that can be either confidential or public. The profiles are:

* Web Application
* User Agent
* Native

### Web Application

A web application is an application running on a web server. In reality, a web application typically consists of both a browser part and a server part. If a web application needs access to a resource server (e.g. to Facebook user accounts), then the client password could be stored on the server. The password would thus be confidential.

Here is an illustration of a confidential client web application:

|  |
| --- |
| Confidential client: Web Application |
| **Confidential client: Web Application.** |

### User Agent Application

A user agent application is for instance a JavaScript application running in a browser. The browser is the user agent. A user agent application may be stored on a web server, but the application is only running in the user agent once downloaded. An example could be a little JavaScript game that only runs in the browser.

Here is an illustration of a client user agent application (Client Profile: Public)

|  |
| --- |
| Public client: User Agent Application |
| **Public client: User Agent Application.** |

### Native Application

A native application is for instance a desktop application or a mobile phone application. Native applications are typically installed on the user’s computer or device (phone, tablet etc.). Thus, the client password will be stored on the user’s computer or device too – hence not confidential/public.

Here is an illustration of a client native application: (Client profile: Public)

|  |
| --- |
| Public client: Native Application |
| **Public client: Native Application.** |

### Hybrid Applications

Some applications are hybrids of these profiles. For instance, a native application can have a server part too, that does part of the work (e.g. store data). The OAuth 2.0 specification says nothing about such hybrids. However, in most cases a hybrid will be able to use the authentication models of one of these profiles.

# OAuth 2.0 Authorization

When a client application wants access to the resources of a resource owner, hosted on a resource server, the client application must first obtain an authorization grant. This text explains how such an application grant is obtained.

**Client ID, Client Secret and Redirect URI**

Before a client application can request access to resources on a resource server, the client application must first register with the authorization server associated with the resource server.

The registration is typically a one-time task. Once registered, the registration remains valid, unless the client app registration is revoked.

At registration the client application is assigned a client ID and a client secret (password) by the authorization server. The client ID and secret is unique to the client application on that authorization server. If a client application registers with multiple authorization servers (e.g. both Facebook, Twitter and Google), each authorization server will issue its own unique client ID to the client application.

Whenever the client application requests access to resources stored on that same resource server, the client application needs to authenticate itself by sending along the client ID and the client secret to the authorization server.

During the registration the client also registers a redirect URI. This redirect URI is used when a resource owner grants authorization to the client application. When a resource owner has successfully authorized the client application via the authorization server, the resource owner is redirected back to the client application, to the redirect URI.

**Authorization Grant**

The ***authorization grant*** is given to a client application by the resource owner, in cooperation with the authorization server associated with the resource server.

The OAuth 2.0 specification lists four different types of authorization grants. Each type has different security characteristics. The authorization grant types are:

* Authorization Code
* Implicit
* Resource Owner Password Credentials
* Client Credentials

Each of these authorization grant types is covered in the following sections.

**Authorization Code**

An authorization grant using an authorization code works like this:

The resource owner (user) accesses the client application. The client application tells the user to login to the client application via an authorization server (e.g. Facebook, Twitter, Google etc.).

To login via the authorization server, the user is redirected to the authorization server by the client application. The client application sends its client ID along to the authorization server, so the authorization server knows which application is trying to access the protected resources.

The user logs in via the authorization server. After successful login the user is asked if she wants to grant access to her resources to the client application. If the user accepts, the user is redirected back to the client application.

When redirected back to the client application, the authorization server sends the user to a specific redirect URI, which the client application has registered with the authorization server ahead of time. Along with the redirection, the authorization server sends an authorization code, representing the authorization.

When the redirect URI in the client application is accessed, the client application connects directly to the authorization server. The client application sends the authorization code along with its own client ID and client secret. If the authorization server can accept these values, the authorization server sends back an access token.

The client application can now use the access token to request resources from the resource server. The access token serves as both authentication of the client, and authorization to access the resources.

To summarize the steps:

* Resource owner accesses client application
* Client application redirects user through an user-agent (like browser) to an authorization server along with the Client Id (through a pre-registration of the client)
* Authorization server authenticates the resource owner and obtains authorization to share the resource with the client application
* Authorization server redirects resource owner to a specific redirect client URI along with an *authorization code*
* The client application connects to the authorization server with the authorization code, client id and client secret
* Authorization server sends back an *access token*
* Client application uses the access token to access the resource on the resource server
* Resource server reaches out to authorization server to validate the token
* Resource server returns the requested resource

Security Implications

* Ability to authenticate the client
* Ability to transmit the access token directly to the client without routing it through the user agent

Usage Scenarios

* Public APIs which can be accessed by any number of clients

Here is a diagram illustrating the authorization process when using authorization code to authorize a client application:

|  |
| --- |
| Authorization grant via authorization code. |
| **Authorization grant via authorization code.** |

**Implicit**

An implicit authorization grant is similar to an authorization code grant, except the access token is returned to the client application already after the user has finished the authorization. The access token is thus returned when the user agent is redirected to the redirect URI.

This of course means that the access token is accessible in the user agent, or native application participating in the implicit authorization grant. The access token is not stored securely on a web server.

Furthermore, the client application can only send its client ID to the authorization server. If the client were to send its client secret too, the client secret would have to be stored in the user agent or native application too. That would make it vulnerable to hacking.

Implicit authorization grant is mostly used in a user agent or native client application. The user agent or native application would receive the access token from the authorization server.

To summarize the steps:

* Resource owner accesses client application
* Client application redirects user through an user-agent (like browser) to an authorization server along with the Client Id (through a pre-registration of the client)
* Authorization server authenticates the resource owner and obtains authorization to share the resource
* Authorization server redirects resource owner to a specific redirect client URI along with an *access token*
* Client application uses the access token and Client Id to access the resource on the resource server
* Resource server reaches out to authorization server to validate the token
* Resource server returns the requested resource

Security Implications

* Cannot authenticate the client as client secret cannot be shared with authorization server without passing it through user agent
* User agent has access to access token

Advantages

* Optimized for clients implemented in a browser using a scripting language (like JavaScript)
* Helps improve responsiveness and efficiency of some clients by reducing the # of roundtrips

Usage Scenarios

* Mobile app or remote location access
* User can be trusted

Here is an illustration of implicit authorization grant:

|  |
| --- |
| Implicit authorization grant. |
| **Implicit authorization grant.** |

**Resource Owner Password Credentials**

The resource owner password credentials authorization grant method works by giving the client application access to the resource owners credentials. For instance, a user could type his Twitter user name and password (credentials) into the client application. The client application could then use the user name and password to access resources in Twitter.

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

Using the resource owner password credentials requires a lot of trust in the client application. You do not want to type your credentials into an application you suspect might abuse it.

The resource owner password credentials would normally be used by user agent client applications, or native client applications.

To summarize the steps:

* Resource owner accesses a trusted client application
* Client application requests for user credentials for an authorization server
* Client application reaches out to authorization server with client Id and user credentials requesting for an access token
* Authorization server authenticates the resource owner through the user credentials provided by the client application and sends back a long-lived *access token*
* Client application stores the access token and not the resource owner credentials
* Client application uses the access token to access the resource on the resource server
* Resource server reaches out to authorization server to validate the token
* Resource server returns the requested resource
* In case the access token expires, client application requests for a *refresh token*

Security Implications

* Client application needs to be highly trusted

Usage Scenarios

* Should only be used when there is a high degree of trust between the resource owner and the client (for e.g. device OS, highly privileged application)
* And when other authorization grant types are not available

**Client Credentials**

Client credential authorization is for the situations where the client application needs to access resources or call functions in the resource servers, which are not related to a specific resource owner (e.g. user). For instance, obtaining a list of venues from Foursquare. This does not necessarily have anything to do with a specific Foursquare user.

# OAuth 2.0 Endpoints

OAuth 2.0 defines a set of endpoints. An endpoint is typically a URI on a web server. For instance, the address of a Java servlet, JSP page, PHP page, ASP.NET page etc.

The endpoints defined are:

* Authorization Endpoint
* Token Endpoint
* Redirection Endpoint

The authorization endpoint and token endpoint are both located on the authorization server. The redirection endpoint is located in the client application. Each of these endpoints are covered below.

The endpoints are illustrated in this diagram:

|  |
| --- |
| OAuth 2.0 Endpoints. |
| **OAuth 2.0 Endpoints.** |

The OAuth 2.0 specification does not describe how the URI of these endpoints are found or documented. That is up to each implementer to decide. Most sites will have a subsite for developers documenting these endpoints.

**Authorization Endpoint**

The authorization endpoint is the endpoint on the authorization server where the resource owner logs in, and grants authorization to the client application.

**Token Endpoint**

The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, client ID and client secret, for an access token.

**Redirect Endpoint**

The redirect endpoint is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint.

# OAuth 2.0 Requests and Responses

When the client application requests authorization and access tokens it sends HTTP requests to the authorization server, to its authorization and token endpoints. What request and response is sent forth and back depends on the authorization grant type. Remember, the four grant types are:

* Authorization Code Grant
* Implicit Grant
* Resource Owner Password Credentials Grant
* Client Credentials Grant

The request and response of each of these authorization grant types is explained in more detail in the following, separate texts.

The information presented in the following texts is, however, mostly a summary. In order to get all the details of what it all means, you may have to consult the OAuth 2.0 specification, or the documentation of the system (Facebook, Twitter, Foursquare etc.) you are trying to integrate with.

# OAuth 2.0 Authorization Code Requests and Responses

The authorization code grant consists of 2 requests and 2 responses in total. An authorization request + response, and a token request + response.

## Authorization Request

The authorization request is sent to the authorization endpoint to obtain an authorization code. Here are the parameters used in the request:

|  |  |
| --- | --- |
| response\_type | Required. Must be set to code |
| client\_id | Required. The client identifier as assigned by the authorization server, when the client was registered. |
| redirect\_uri | Optional. The redirect URI registered by the client. |
| scope | Optional. The possible scope of the request. |
| state | Optional (recommended). Any client state that needs to be passed on to the client request URI. |

## Authorization Response

The authorization response contains the authorization code needed to obtain an access token. Here are the parameters included in the response:

|  |  |
| --- | --- |
| code | Required. The authorization code. |
| state | Required, if present in request. The same value as sent by the client in the state parameter, if any. |

## Authorization Error Response

If an error occurs during authorization, two situations can occur.

The first is, that the client is not authenticated or recognized. For instance, a wrong redirect URI was sent in the request. In that case the authorization server must not redirect the resource owner to the redirect URI. Instead it should inform the resource owner of the error.

The second situation is that client is authenticated correctly, but that something else failed. In that case the following error response is sent to the client, included in the redirect URI:

|  |  |
| --- | --- |
| error | Required. Must be one of a set of predefined error codes. See the specification for the codes and their meaning. |
| error\_description | Optional. A human-readable UTF-8 encoded text describing the error. Intended for a developer, not an end user. |
| error\_uri | Optional. A URI pointing to a human-readable web page with information about the error. |
| state | Required, if present in authorization request. The same value as sent in the state parameter in the request. |

## Token Request

Once an authorization code is obtained, the client can use that code to obtain an access token. Here is the access token request parameters. And it is a POST call.

|  |  |
| --- | --- |
| grant\_type | Required. Must be set to authorization\_code . |
| code | Required. The authorization code received by the authorization server. |
| redirect\_uri | Required, if the request URI was included in the authorization request. Must be identical then. |
| client\_id | Required, if client is not authenticating itself with the Authorization Server |

## Token Response

The response to the access token request is a JSON string containing the access token plus some more information:

{ "access\_token" : "...",

"token\_type" : "...",

"expires\_in" : "...",

"refresh\_token" : "...",

}

The access\_token property is the access token as assigned by the authorization server.

The token\_type property is a type of token assigned by the authorization server (e.g. bearer, HoK, Mac)

The expires\_in property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

The refresh\_token property contains a refresh token in case the access token can expire. The refresh token is used to obtain a new access token once the one returned in this response is no longer valid.

# OAuth 2.0 Implicit Requests and Responses

The implicit grant consists of only 1 request and 1 response.

### Implicit Grant Request

The implicit grant request contains the following parameters:

|  |  |
| --- | --- |
| response\_type | Required. Must be set to token . |
| client\_id | Required. The client identifier as assigned by the authorization server, when the client was registered. |
| redirect\_uri | Optional. The redirect URI registered by the client. |
| scope | Optional. The possible scope of the request. |
| state | Optional (recommended). Any client state that needs to be passed on to the client request URI. |

### Implicit Grant Response

The implicit grant response contains the following parameters. Note, that the implicit grant response is not JSON.

|  |  |
| --- | --- |
| access\_token | Required. The access token assigned by the authorization server. |
| token\_type | Required. The type of the token |
| expires\_in | Recommended. A number of seconds after which the access token expires. |
| scope | Optional. The scope of the access token. |
| state | Required, if present in the autorization request. Must be same value as state parameter in request. |

### Implicit Grant Error Response

If an error occurs during authorization, two situations can occur.

The first is, that the client is not authenticated or recognized. For instance, a wrong redirect URI was sent in the request. In that case the authorization server must not redirect the resource owner to the redirect URI. Instead it should inform the resource owner of the error.

The second situation is that client is okay, but that something else happened. In that case the following error response is sent to the client, included in the redirect URI:

|  |  |
| --- | --- |
| error | Required. Must be one of a set of predefined error codes. See the specification for the codes and their meaning. |
| error\_description | Optional. A human-readable UTF-8 encoded text describing the error. Intended for a developer, not an end user. |
| error\_uri | Optional. A URI pointing to a human-readable web page with information about the error. |
| state | Required, if present in authorization request. The same value as sent in the state parameter in the request. |

# OAuth 2.0 Resource Owner Password Credentials Grant - Requests and Response

The resource owner password credentials authorization contains a single request + response.

### Resource Owner Password Credentials Grant Request

The request contains the following parameters:

|  |  |
| --- | --- |
| grant\_type | Required. Must be set to password |
| username | Required. The username of the resource owner, UTF-8 encoded. |
| password | Required. The password of the resource owner, UTF-8 encoded. |
| scope | Optional. The scope of the authorization. |

### Resource Owner Password Credentials Grant Response

The response is a JSON structure containing the access token. The JSON structure looks like this:

{ "access\_token" : "...",

"token\_type" : "...",

"expires\_in" : "...",

"refresh\_token" : "...",

}

The access\_token property is the access token as assigned by the authorization server.

The token\_type property is a type of token assigned by the authorization server.

The expires\_in property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

The refresh\_token property contains a refresh token in case the access token can expire. The refresh token is used to obtain a new access token once the one returned in this response is no longer valid.

# OAuth 2.0 Client Credentials Grant - Requests and Response

The client credentials grant request contains the following parameters:

|  |  |
| --- | --- |
| grant\_type | Required. Must be set to client\_credentials . |
| scope | Optional. The scope of the authorization. |

### Client Credentials Grant Response

The client credentials response contains the following parameters:

{ "access\_token" : "...",

"token\_type" : "...",

"expires\_in" : "...",

}

The access\_token property is the access token as assigned by the authorization server.

The token\_type property is a type of token assigned by the authorization server.

The expires\_in property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

A refresh token should not be included for this type of authorization request.

# OAuth 2.0 Implementations

### Implementations from <http://oauth.net/2/>

#### Server Libraries

* Java
  + [MitreID (with OpenID Connect)](https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server)
  + [Apache Oltu](http://oltu.apache.org/)
  + [Spring Security for OAuth](http://static.springsource.org/spring-security/oauth/)
  + [Apis Authorization Server (v2-31)](https://github.com/OpenConextApps/apis)
  + [Restlet Framework (draft 30)](http://www.restlet.org/)
  + [Apache CXF](http://cxf.apache.org/)
* PHP
  + [PHP OAuth2 Server](https://github.com/bshaffer/oauth2-server-php) and [Demo](https://github.com/bshaffer/oauth2-demo-php)
  + [PHP OAuth 2.0 Auth and Resource Server](https://github.com/thephpleague/oauth2-server) and [Demo](https://github.com/lncd/oauth2-example-auth-server)
  + [PHP OAuth 2.0](https://github.com/fkooman/php-oauth) (AS with SAML/BrowserID AuthN, with management REST API)
  + [PHP OAuth2.0](https://github.com/authbucket/oauth2) for [Silex](http://silex.sensiolabs.org/) and [Demo](http://oauth2.authbucket.com/demo)
  + [PHP OAuth2.0](https://github.com/authbucket/oauth2-bundle) for [Symfony](http://symfony.com/) and [Demo](http://oauth2-bundle.authbucket.com/demo)
* Python
  + [Python OAuth 2.0 Client + Server Library](https://github.com/NateFerrero/oauth2lib)
  + [OAuthLib](https://github.com/idan/oauthlib) (a generic implementation of the OAuth request-signing logic) is avaliable for [Django](https://github.com/evonove/django-oauth-toolkit) and [Flask](https://github.com/lepture/flask-oauthlib) web frameworks
* NodeJS
  + [NodeJS OAuth 2.0 Provider](https://github.com/t1msh/node-oauth20-provider)
  + [Mozilla Firefox Accounts](https://github.com/mozilla/?query=fxa). A full stack Identy Provider system developed to support Firefox market place and other services
* Ruby
  + [Ruby OAuth2 Server (draft 18)](https://github.com/nov/rack-oauth2)
* .NET
  + [.NET DotNetOpenAuth](http://www.dotnetopenauth.net/)
  + [Thinktecture IdentityServer](https://github.com/thinktecture/Thinktecture.IdentityServer.v3)
* Erlang
  + [Erlang Oauth2 Server framework](https://github.com/kivra/oauth2)

#### Proxy services

* [OAuth.io](https://github.com/oauth-io) (self hosted), and also you can use as [an external service](https://oauth.io/)

#### Client Libraries

* PHP
  + [league/oauth2-client](https://github.com/thephpleague/oauth2-client): OAuth 2.0 Client from [the League of Extraordinary Packages](https://thephpleague.com/)
  + [oauth-api](http://www.phpclasses.org/package/7700-PHP-Authorize-and-access-APIs-using-OAuth.html) from [PHP Classes](http://www.phpclasses.org/)
  + [PHP OAuth 2.0 Authorization Code Grant Client](https://github.com/fkooman/php-oauth-client)
  + [OAuth2/OpenID Connect Client Library for PHP/Zend Framework 2](https://github.com/ivan-novakov/php-openid-connect-client)
* Objective C
  + [Cocoa](http://github.com/leebyron/cocoa-oauth2)
  + [iPhone and iPad](http://github.com/lukeredpath/LROAuth2Client)
  + [iOS and Mac OS X (draft 10)](http://github.com/nxtbgthng/OAuth2Client)
* Swift
  + [OAuthSwift](https://github.com/dongri/OAuthSwift)
* Java
  + [Apache Oltu](http://oltu.apache.org/)
  + [Spring Social](http://www.springsource.org/spring-social)
  + [Spring Security for OAuth](http://static.springsource.org/spring-security/oauth/)
  + [Restlet Framework (draft 30)](http://www.restlet.org/)
  + [ScribeJava](https://github.com/scribejava/scribejava)
* Scala
  + [Silhouette](http://silhouette.mohiva.com/)
* Python
  + [sanction](http://github.com/demianbrecht/sanction)
  + [rauth](http://github.com/litl/rauth)
  + [Authomatic](http://peterhudec.github.io/authomatic/)
* Ruby
  + [Ruby Gem](http://github.com/intridea/oauth2)
  + [Ruby](http://github.com/aflatter/oauth2-ruby)
* Javascript
  + [Javascript](http://github.com/andreassolberg/jso)
* .NET
  + [OWIN Middleware](http://www.nuget.org/packages/Microsoft.Owin.Security.OAuth)
  + [DotNetOpenAuth](http://www.dotnetopenauth.net/)
  + [Spring Social for .NET](http://www.springframework.net/social/)
* Qt/C++
  + [O2 (supports OAuth 1.0a and 2.0)](https://github.com/pipacs/o2)
* Lua/Corona SDK
  + [Corona/Lua OAuth 2.0 API](http://selz.co/1kxjJVl)
* Dart
  + [Dart OAuth 2.0 Client](https://pub.dartlang.org/packages/oauth2)

OAuth Service provider

* Authlete (OAuth 2.0 and OIDC provider on Cloud)
* Auth0