**OAuth Authentication:**

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

Here is a diagram illustrating the concept:

|  |
| --- |
| 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 Vs OAuth 1.0**

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

**Top Differences between OAuth 1.0 and OAuth 2.0 for API Calls**

OAuth 1.0 (the current spec version is [1.0a](http://oauth.net/core/1.0a/), which fixes a security problem with 1.0) solves an important problem in the world of APIs -- how one web application can give another application API access without requiring that the user give out their password. OAuth 1.0 solves this problem in a clever way through a secure handshake, via API calls, between the two applications. This has allowed APIs to go in places where they could never go before.

OAuth 1.0 works by ensuring that the API client and server share a token, which is like a username, and a token secret, which is like a password. The client must generate a "signature" on every API call by encrypting a bunch of unique information using the token secret. The server must generate the same signature, and only grant access if both signatures match.

The advantage of this approach is that there is no way to find out the token secret, because it is always encrypted when it's sent over the network, and only the client and server have the keys. It doesn't matter if the data is eavesdropped on a WiFi network in Starbucks or captured by a proxy like [Apigee](http://apigee.com/) -- there's no way to see the secret, so there's no way to impersonate the client based on what's sent over the network. All of this is done without requiring SSL, since SSL can slow down the client and server alike, and make deployment of the API server more complex.

However, both client and server developers found the complexity of generating and validating signatures to be too much. There are many tricky things that a developer must get right, down to exactly what type of "URL Encoding" is used (it's not exactly the same as it's used in other places). If the client or server makes a single tiny mistake in the signature, it's invalid and it's hard to figure out what went wrong.

OAuth 2.0 promises to simplify this stuff in a number of ways:

**1.** SSL is required for all the communications required to generate the token. This is a huge decrease in complexity because those complex signatures are no longer required.  
**2.** Signatures are not required for the actual API calls once the token has been generated -- SSL is also strongly recommended here.  
**3.** Once the token was generated, OAuth 1.0 required that the client send two security tokens on every API call, and use both to generate the signature. OAuth 2.0 has only one security token, and no signature is required.  
**4.** It is clearly specified which parts of the protocol are implemented by the "resource owner," which is the actual server that implements the API, and which parts may be implemented by a separate "authorization server." That will make it easier for products like Apigee to offer OAuth 2.0 support to existing APIs.

For these reasons, OAuth 2.0 has already been adopted by companies like Facebook, which uses the draft spec in its [Graph API](http://app.apigee.com/console/facebook). Of course, it's a new spec, which means there are new requirements and use cases that make it more complex. For instance, OAuth 2.0 also clearly lays out how to use OAuth entirely inside a browser using JavaScript that has no way to securely store a token, and it explains at a high level how to use it on a mobile phone or even on a device that has no web browser at all.

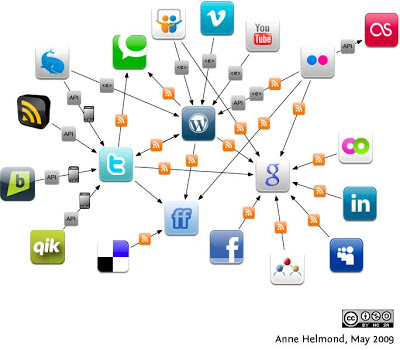
Finally, although the developers of the world will not miss generating OAuth 1.0 signatures, they served a purpose, because they allowed a client to send its token and secret securely to a server without requiring SSL. For APIs and devices that do not want to support SSL for performance or complexity reasons, signatures are still a good choice. Right now, signatures have been removed from the OAuth 2.0 spec, but they'll be added to a separate extension spec at some point.

So should you use OAuth 2.0 today? Here are a few things to consider:

* **Spec changes**. OAuth 2.0 has not reached a stable IETF draft yet. If you implement it today, are you prepared to change your implementation every few weeks until the committee has agreed on a stable version? OAuth 1.0a, on the other hand, is already a well-defined standard that's not going to change any time soon.
* **Implementations.** There are a number of code libraries for both client and server that support 1.0a today. There aren't as many for 2.0, so you're going to have to build more stuff on your own.
* **Complexity.** There's no doubt that 2.0 is easier to implement both on the client and server side.
* **Performance.** If you are unwilling or unable to use SSL for all of your API traffic, then OAuth 2.0 is not a good choice until some sort of signature extension is added to the spec. OAuth 1.0a already supports signatures, which are complex but allow you to securely exchange tokens without requiring the use of SSL.

**OAuth :**

If you use social media (e.g. facebook, twitter) it's likely that you've used OAuth. OAuth works also for devices, e.g. my TV uses OAuth to let me access my Youtube account.

[](http://3.bp.blogspot.com/-hBQZ68-Fo2Y/UOq79nCqKcI/AAAAAAAAAXs/yBDVNR27Jlc/s1600/social-media-social-networking-connections.jpg)

### Usages:

* **Delegated authorization** e.g. a (web) application (the client / resource **consumer**) want to access a user data  in a resource **provider** (e.g. google calendar or facebook photos/friends), the application rely on user permission (authorisation) using an**authorization server** (e.g. facebook, google)  to get access to user's data.
* OpenID, which is build on top of OAuth protocol, provides single sign on / (federated)**delegated authentication**.

### Advantages:

* To **avoid too many password accounts** for different applications that will increase the phishing risk, ear drop risk and revocation problem (you need to cancel/change all the password accounts in all applications)
* To **avoid sharing password to resource providers**, only the security provider (e.g. google) registers your password
* To **standardize** different (delegated) authorization methods from different vendors. At this moment in practice there are still minor differences between different vendor implementations, but the distances between them are getting smaller and generic APIs (e.g. Scribe) that adapt to different vendors are available.

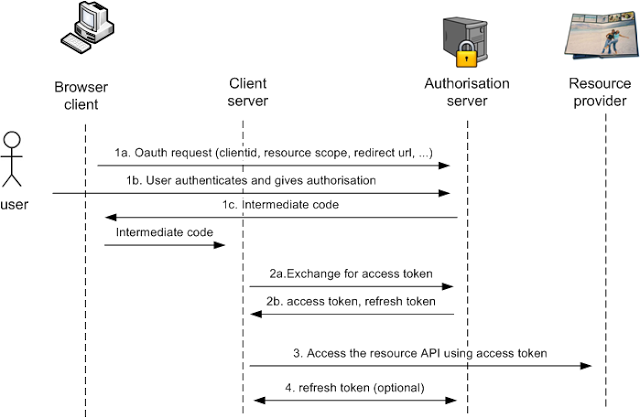
### Best practice

* the auth token is sent with **http header** instead of URL GET parameters: less obvious, to avoid the token being cached/logged as url entries
* use **SSL**to avoid the attackers steal the token
* **limit** the token **lifespan** (if you revoke the token it will not last too long)
* use random **salt** to prevent CSRF attack.
* manage user experience**: tell user**what happens "e.g. This application needs your approval to use this resource. We will redirect you to an authorisation server (e.g. google) so that you can give permission."

### Protocol:  the web server application flow (a.k.a. the 3 legged flow)

This is the strongest OAuth protocol, to avoid the client browser (e.g. malicious javascript) to catch the access token by using **intermediately** **autho code** (which later is being exchanged for an access token via server to server call that safer than browser to server call).

First the client application has to register to the autho server / oauth security provider (e.g. google) to establish clientID & password.

The flow:  
[](http://4.bp.blogspot.com/-1yCiD2wddmA/UOq6-05e8lI/AAAAAAAAAXg/SngmJuajRkY/s1600/oauth.png)

1. The application makes **oauth request**to the the authorisation server with parameters: clientid (from the client application registration), resorce scope (e.g. google calendar), redirect url (where the user will be returned after they approve the access) and other parameters such as state (random salt against CSRF) and access\_type ('offline' for the 3 legged flow). **The user has to login and then gives permission/authorisation** to the application via authorisation server web interface. The authorisation server then return the user (& send the**intermediate code**) to the redirect url.

2. The client application then can **exchange the intermediate code for an access token**.

3. The application then can **access the user data** in the resource provider using**access token**. The resource provider will validate the access token. The validation process is not defined in the standard, but for example some OAuth implementation provide API in the OAuth server where the resource provider can exchange the token with user's id & scope information and then compare this result with the resource provider's own ACL.

4. Optional: in some scenarios the access token will be expired, the application need to obtain a new access token using **refresh token**.

### Other protocols:

* **client** flow (a.k.a. the 2 legged flow) : the client received the access token directly (instead of have to exchange the intermediate code.) This approach is less secure (since the access token is exposed to  the client browser, so for example a malicious javascript can steal it) but good enough for many non critical applications (e.g. invitations for photos viewer).
* **password** flow: use password to be exchanged with an token. This approach is less secure (the application/resource provider will know your password) but can be used for login to trusted applications inside a company (e.g. webmail, CRM).
* **client credential** flow: use the client credential from the client registration. This typically is used for internal applications or trusted cloud integration e.g. salesforce CRM to your company.
* **devices** flow: for mobile devices, smartphones, TV.