OAuth 2 Simplified

Authorization

This post describes OAuth 2.0 in a simplified format to help developers and service providers implement the protocol.

The OAuth 2 spec can be a bit confusing to read, so I've written this post to help describe the terminology in a simplified format. The core spec leaves many decisions up to the implementer, often based on security tradeoffs of the implementation. Instead of describing all possible decisions that need to be made to successfully implement OAuth 2, this post makes decisions that are appropriate for most implementations.

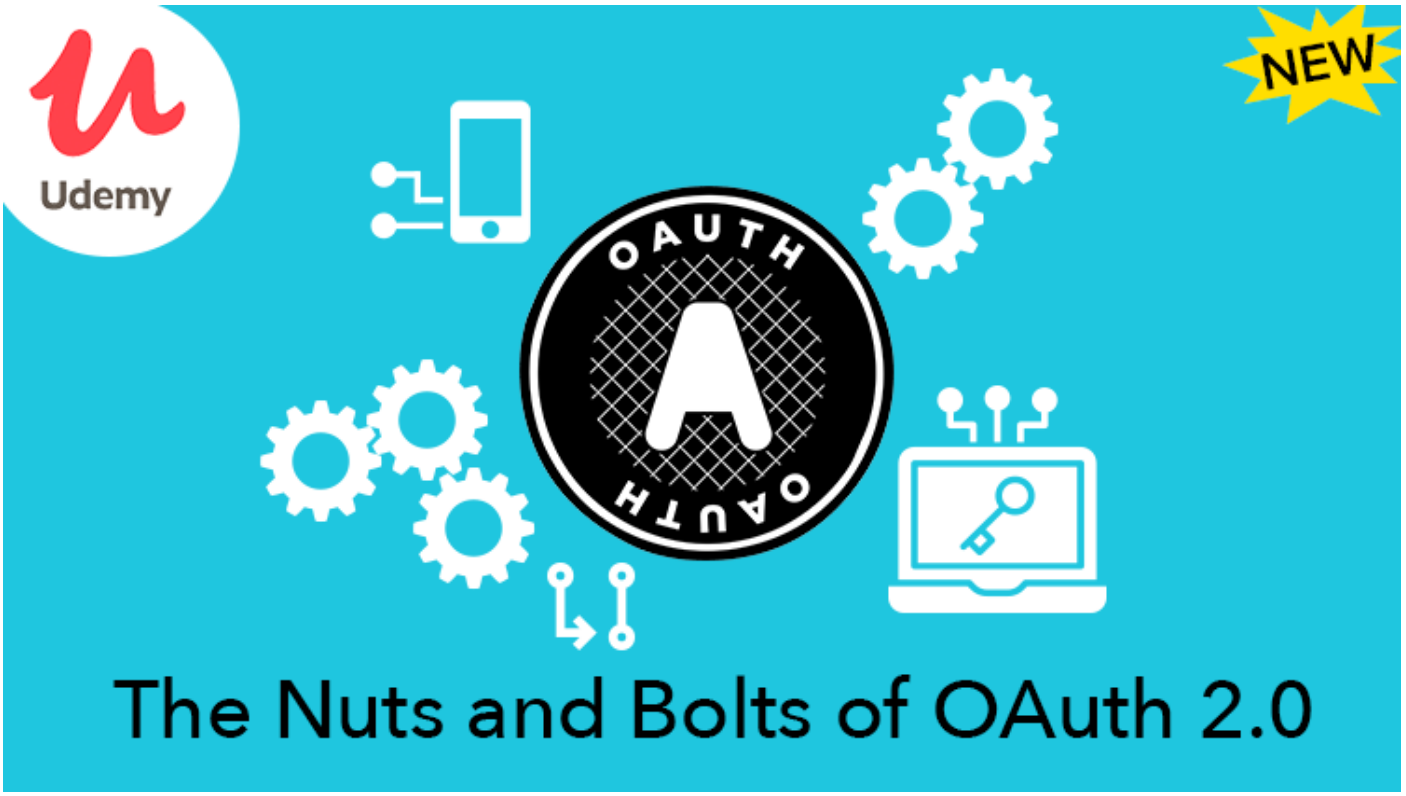


Table of Contents

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 Authorization Server

This is the server that presents the interface where the user approves or denies the request. In smaller implementations, this may be the same server as the API server, but larger scale deployments will often build this as a separate component.

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 served via HTTPS. This helps prevent tokens from being intercepted during the authorization process. Native apps may register a redirect URI with a custom URL scheme for the application, which may look like `demoapp://redirect`.

Client ID and Secret

After registering your app, you will receive a client ID and optionally 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 single-page Javascript apps or native apps, then the secret is not used, and ideally the service shouldn't issue a secret to these types of apps in the first place.

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, browser-based and mobile apps

Password for logging in with a username and password (only for first-party apps)

Client credentials for application access without a user present

Implicit was previously recommended for clients without a secret, but has been superseded by using the Authorization Code grant with PKCE.

Each use case is described in detail below.

Web Server Apps

Web server apps are the most common type of application you encounter when dealing with OAuth servers. Web apps are written in a server-side language and run on a server where the source code of the application is not available to the public. This means the application is able to use its client secret when communicating with the authorization server, which can help avoid many attack vectors.

Authorization

Create a "Log In" link sending the user to:

```
https://authorization-server.com/auth?response_type=code&
  client_id=CLIENT_ID&redirect_uri=REDIRECT_URI&scope=photos&
state=1234zyx
```

response_type=code - Indicates that your server expects to receive an authorization code

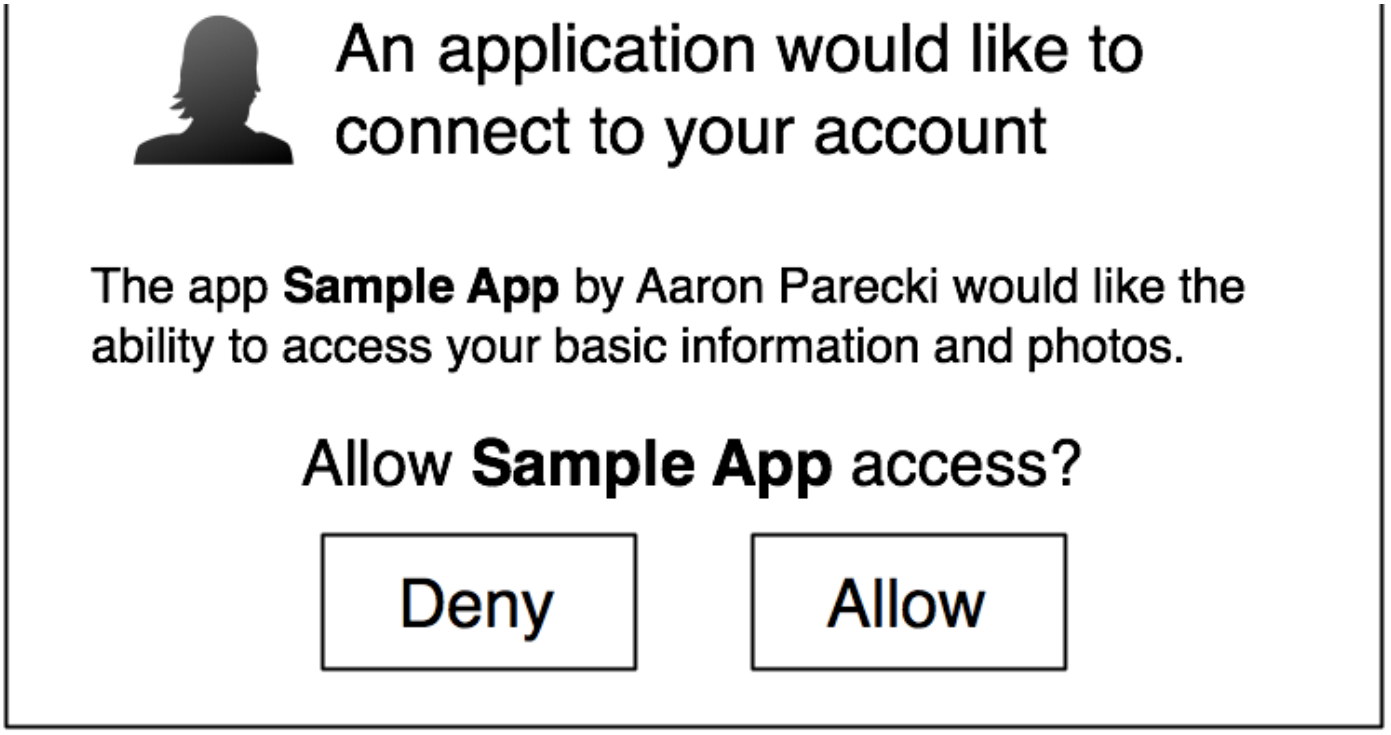
client_id - The client ID you received when you first created the application

redirect_uri - Indicates the URI to return the user to after authorization is complete

scope - One or more scope values indicating which parts of the user's account you wish to access

state - A random string generated by your application, which you'll verify later

The user sees the authorization prompt



If the user clicks "Allow," the service redirects the user back to your site with an authorization code

```
https://example-app.com/cb?code=AUTH_CODE_HERE&state=1234zyx
```

code - The server returns the authorization code in the query string

state - The server returns the same state value that you passed

You should first compare this state value to ensure it matches the one you started with. You can typically store the state value in a cookie or session, and compare it when the user comes back. This helps ensure your redirection endpoint isn't able to be tricked into attempting to exchange arbitrary authorization codes.

Getting an Access Token

Your server exchanges the authorization code for an access token by making a POST request to the authorization server's token endpoint:

```
POST https://api.authorization-server.com/token
  grant_type=authorization_code&
  code=AUTH_CODE_HERE&
  redirect_uri=REDIRECT_URI&
  client_id=CLIENT_ID&
  client_secret=CLIENT_SECRET
```

grant_type=authorization_code - The grant type for this flow is authorization_code

code=AUTH_CODE_HERE - This is the code you received in the query string

redirect_uri=REDIRECT_URI - Must be identical to the redirect URI provided in the original link

client_id=CLIENT_ID - The client ID you received when you first created the application

client_secret=CLIENT_SECRET - Since this request is made from server-side code, the secret is included

The server replies with an access token and expiration time

```
{
  "access_token":"RsT5OjbzRn430zqMLgV3Ia",
  "expires_in":3600
}
```

or if there was an error

```
{
  "error":"invalid_request"
}
```

Security: Note that the service must require apps to pre-register their redirect URIs.

Single-Page Apps

Single-page apps (or browser-based apps) run entirely in the browser after loading the source code from a web page. Since the entire source code is available to the browser, they cannot maintain the confidentiality of a client secret, so the secret is not used in this case. The flow is based on the authorization code flow above, but with the addition of a dynamically generated secret used on each request. This is known as the PKCE extension.

Authorization

Create a random string between 43-128 characters long, then generate the url-safe base64-encoded SHA256 hash of the string. The original random string is known as the `code_verifier`, and the hashed version is known as the `code_challenge`.

Create a random string (code verifier), e.g. `5d2309e5bb73b864f989753887fe52f79ce5270395e25862da6940d5`

Create the SHA256 hash, then base64-encode the string (code challenge): `MChCW5vD-3h03HMGFZYskOSTir7II_MMTb8a9rJNhnI`

(You can use the helper utility at example-app.com/pkce to generate a secret and hash.)

Create a "Log In" link like the authorization code flow above, but now include the code challenge in the request:

```
https://authorization-server.com/auth?response_type=code&
  client_id=CLIENT_ID&redirect_uri=REDIRECT_URI&scope=photos&
state=1234zyx&code_challenge=CODE_CHALLENGE&code_challenge_method=S256
```

response_type=code - Indicates that your server expects to receive an authorization code

client_id - The client ID you received when you first created the application

redirect_uri - Indicates the URI to return the user to after authorization is complete

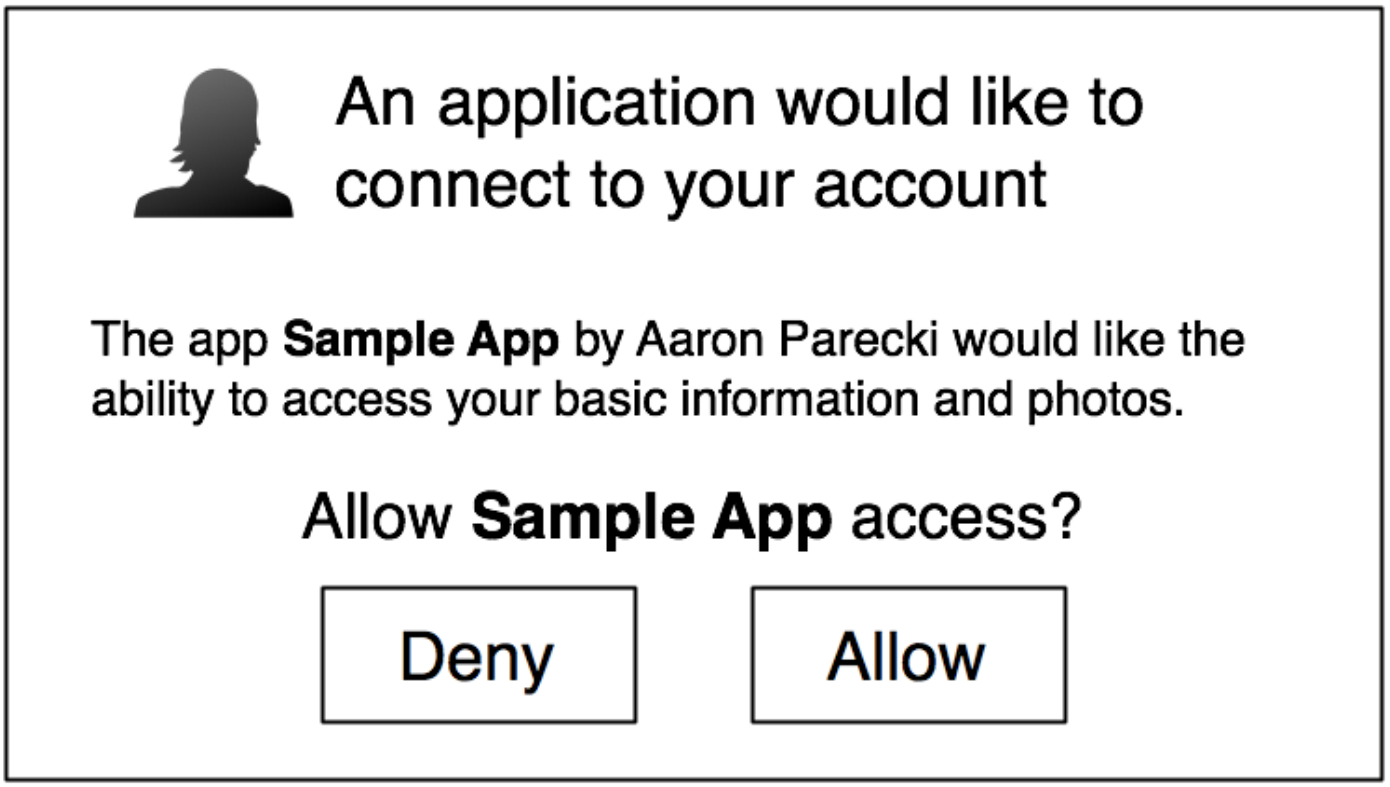
scope - One or more scope values indicating which parts of the user's account you wish to access

state - A random string generated by your application, which you'll verify later

code_challenge - The URL-safe base64-encoded SHA256 hash of the secret

code_challenge_method=S256 - Indicate which hashing method you used (S256)

The user sees the authorization prompt



If the user clicks "Allow," the service redirects the user back to your site with an auth code

```
https://example-app.com/cb?code=AUTH_CODE_HERE&state=1234zyx
```

code - The server returns the authorization code in the query string

state - The server returns the same state value that you passed

You should first compare this state value to ensure it matches the one you started with. You can typically store the state value in a cookie, and compare it when the user comes back. This ensures your redirection endpoint isn't able to be tricked into attempting to exchange arbitrary authorization codes.

You can find a complete example of using PKCE in JavaScript in my blog post Is the OAuth Implicit Flow Dead?

Getting an Access Token

Now you'll need to exchange the authorization code for an access token, but instead of providing a pre-registered client secret, you send the PKCE secret you generated at the beginning of the flow.

```
POST https://api.authorization-server.com/token
  grant_type=authorization_code&
  code=AUTH_CODE_HERE&
  redirect_uri=REDIRECT_URI&
  client_id=CLIENT_ID&
  code_verifier=CODE_VERIFIER
```

grant_type=authorization_code - The grant type for this flow is authorization_code

code=AUTH_CODE_HERE - This is the code you received in the query string

redirect_uri=REDIRECT_URI - Must be identical to the redirect URI provided in the original link

client_id=CLIENT_ID - The client ID you received when you first created the application

code_verifier=CODE_VERIFIER - The random secret you generated at the beginning

The authorization server will hash the verifier and compare it to the challenge sent in the request, and only issue the access token if they match. This ensures that even if someone was able to intercept the authorization code, they will not be able to use it to get an access token since they won't have the secret.

Mobile Apps

Like browser-based apps, mobile apps also cannot maintain the confidentiality of a client secret. Because of this, mobile apps also use the PKCE flow which does not require a client secret. There are some additional concerns that mobile apps should keep in mind to ensure the security of the OAuth flow.

Authorization

Create a "Log in" button sending the user to either the native app of the service on the phone, or a mobile web page for the service. Apps can register a custom URI scheme such as "example-app://" so the native app is launched whenever a URL with that protocol is visited, or they can register URL patterns which will launch the native app if a URL matching the pattern is visited.

Using the Service's Native App

If the user has the native Facebook app installed, direct them to the following URL:

```
fbauth2://authorize?response_type=code&client_id=CLIENT_ID
  &redirect_uri=REDIRECT_URI&scope=email&state=1234zyx
```

response_type=code - indicates that your server expects to receive an authorization code

client_id=CLIENT_ID - The client ID you received when you first created the application

redirect_uri=REDIRECT_URI - Indicates the URI to return the user to after authorization is complete, such as `fb00000000://authorize`

scope=email - One or more scope values indicating which parts of the user's account you wish to access

state=1234zyx - A random string generated by your application, which you'll verify later

For servers that support the PKCE extension (and if you're building a server, you should support the PKCE extension), you'll also include the following parameters. First, create a "code verifier" which is a random string that the app stores locally.

code_challenge=XXXXXXX - This is a base64-encoded version of the sha256 hash of the code verifier string

code_challenge_method=S256 - Indicates the hashing method used to compute the challenge, in this case, sha256.

Note that your redirect URI will probably look like `fb00000000://authorize` where the protocol is a custom URL scheme that your app has registered with the OS.

Using a Web Browser

If the service does not have a native application, you can launch a mobile browser to the standard web authorization URL. Note that you should never use an embedded web view in your own application, as this provides the user no guarantee that they are actually are entering their password in the service's website rather than a phishing site.

You should either launch the native mobile browser, or use the new iOS "SafariViewController" to launch an embedded browser in your application. This API was added in iOS 9, and provides a mechanism to launch a browser inside the application that both shows the address bar so the user can confirm they're on the correct website, and also shares cookies with the real Safari browser. It also prevents the application from inspecting and modifying the contents of the browser, so can be considered secure.

```
https://facebook.com/dialog/oauth?response_type=code&client_id=CLIENT_ID
  &redirect_uri=REDIRECT_URI&scope=email&state=1234zyx
```

Again, if the service supports PKCE, then those parameters should be included as well as described above.

response_type=code - indicates that your server expects to receive an authorization code

client_id=CLIENT_ID - The client ID you received when you first created the application

redirect_uri=REDIRECT_URI - Indicates the URI to return the user to after authorization is complete, such as `fb00000000://authorize`

scope=email - One or more scope values indicating which parts of the user's account you wish to access

state=1234zyx - A random string generated by your application, which you'll verify later

The user will see the authorization prompt

Getting an Access Token

After clicking "Approve", the user will be redirected back to your application with a URL like

```
fb00000000://authorize?code=AUTHORIZATION_CODE&state=1234zyx
```

Your mobile application should first verify that the state corresponds to the state that was used in the initial request, and can then exchange the authorization code for an access token.

The token exchange will look the same as exchanging the code in the web server app case, except that the secret is not sent. If the server supports PKCE, then you will need to include an additional parameter as described below.

```
POST https://api.authorization-server.com/token
  grant_type=authorization_code&
  code=AUTH_CODE_HERE&
  redirect_uri=REDIRECT_URI&
```

```
  client_id=CLIENT_ID&
  code_verifier=VERIFIER_STRING
```

grant_type=authorization_code - The grant type for this flow is authorization_code

code=AUTH_CODE_HERE - This is the code you received in the query string

redirect_uri=REDIRECT_URI - Must be identical to the redirect URI provided in the original link

client_id=CLIENT_ID - The client ID you received when you first created the application

code_verifier=VERIFIER_STRING - The plaintext string that you previously hashed to create the code_challenge

The authorization server will verify this request and return an access token.

If the server supports PKCE, then the authorization server will recognize that this code was generated with a code challenge, and will hash the provided plaintext and confirm that the hashed version corresponds with the hashed string that was sent in the initial authorization request. This ensures the security of using the authorization code flow with clients that don't support a secret.

Other Grant 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 must 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.authorization-server.com/token
  grant_type=password&
  username=USERNAME&
  password=PASSWORD&
  client_id=CLIENT_ID
```

grant_type=password - The grant type for this flow is password

username=USERNAME - The user's username as collected by the application

password=PASSWORD - The user's password as collected by the application

client_id=CLIENT_ID - The client ID you received when you first created the application

The server replies with an access token in the same format as the other grant types.

Note, the client secret is not included here under the assumption that most of the use cases for password

grants will be mobile or desktop apps, where the secret cannot be protected.

Application access

In some cases, applications may need an access token to act on behalf of themselves rather than a user. For example, the service may provide a way for the application to update their own information such as their website URL or 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.authorization-server.com/token
    grant_type=client_credentials&
    client_id=CLIENT_ID&
    client_secret=CLIENT_SECRET
```

The response will include an access token in the same format as the other grant types.

Making Authenticated Requests

The end result of all the grant types is obtaining an access token.

Now that you have an access token, you can make requests to the API. You can quickly make an API request using cURL as follows:

```
curl -H "Authorization: Bearer RsT5OjbzRn430zqMLgV3Ia" \
https://api.authorization-server.com/1/me
```

That's it! Make sure you always send requests over HTTPS and never ignore invalid certificates. HTTPS is the only thing protecting requests from being intercepted or modified.

Differences from OAuth 1.0

OAuth 1.0 was largely based on existing proprietary protocols such as Flickr's "FlickrAuth" and Google's "AuthSub". The result represented the best solution based on actual implementation experience. However, after several years of working with the protocol, the community learned enough to rethink and improve the protocol in three main areas where OAuth 1.0 proved limited or confusing.

You can read more about this in detail in my book OAuth 2.0 Simplified.

Resources

OAuth 2.0 Simplified - the book oauth2simplified.com

Learn more about creating OAuth 2.0 Servers

PKCE Extension

Recommendations for Native Apps

More information is available on OAuth.net

Some content adapted from hueniverse.com.

Previous versions of this post:

July 2012