# Authentication and OAuth

Learn about the importance of authentication and common OAuth flows

# Authentication

## Introduction

Authentication is the process used by applications to determine and confirm identities of users. It ensures that the correct content is shown to users. More importantly, it ensures that incorrect content is secured and unavailable to unauthorized users.

In this article, we’ll discuss a few of the common design patterns for these interactions. You’ll need to have some basic understanding of HTTP requests, since these methods all use HTTP requests to exchange information.

## Password Authentication

The most common implementation of authentication requires a user to input their username or email and a password. The application's server then checks the supplied credentials to determine if the user exists and if the supplied password is correct. If the credentials are correct, the user is logged in and able to use the application as thatthe user.

Typically, upon a successful login, the application will respond with an authentication token (or auth token) for the client to use for additional HTTP requests. This token is then stored on the user's computer, preventing the need for users to continuously log in.

This token generally expires after a certain amount of time, ensuring the correct user is using the application over time as well.

## API Keys

While it is common to think of authentication as the interaction between a human user and an application, sometimes the user is another application.

Many apps expose interfaces to their information in the form of an API (application program interface). For example, the [Spotify API](https://developer.spotify.com/web-api/) provides endpoints for almost all of its functionality. This allows applications to fetch data from the Spotify music catalog and manage user’s playlists and saved music.

Since these external requests could overwhelm a service and also access user information, they need to be secured using authentication.

The most basic pattern for API access from another application is using an API key.

Public APIs usually provide a developer portal where you can register your application and generate a corresponding API key. This key is then unique to your application. When your application makes a request, this key is sent along with it. The API can then verify that your application is allowed access and provide the correct response based on the permission level of your application.

The API can track what type and frequency of requests each application is making. This data can be used to [throttle requests](https://en.wikipedia.org/wiki/Throttling_process_(computing)) from a specific application to a pre-defined level of service. This prevents applications from spamming an endpoint or abusing user data, since the API can easily block that application's API key and prevent further malicious use of the API by that application.

## OAuth

For many applications, a generic developer-level API key is not sufficient. As mentioned earlier, APIs sometimes have the ability to provide access to user-level data. However, most services only provide this private data if the user enables it.

For example, Facebook doesn't want Tinder to access all of their users' data, just the users who have opted in to allowing the sharing of data to better help them find a match in their area.

A basic approach to this problem might be to have the user provide their login credentials to the intermediary application, but this is not very secure and would give full access to the requesting application, when the requesting application might only need a very limited set of privileges to function.

**OAuth** defines a more elegant approach to this problem. It was developed in November 2006 by lead Twitter developer Blaine Cook and version 1.0 was published in April 2010.

OAuth is an **open standard** and is commonly used to grant permission for applications to access user information without forcing users to give away their passwords.

An open standard is a publicly available definition of how some functionality should work. However, the standard does not actually build out that functionality.

As a result, each API is required to implement their own version of OAuth and therefore may have a slightly different implementation or flow. However, they're all based around the same OAuth specification.

This can make using a new OAuth API a little more frustrating. However, with time you will begin noticing the similarities between API authentication flows and be able to use them in your applications with increasing ease. Below is a summary of the standard OAuth flow.

## Generic OAuth Flow

Many applications implementing OAuth will first ask the user to select which service they would like to use for credentials:

authenticationLogin

After selecting the service, the user will be redirected to the service to login. This login confirms the user’s identity and typically provides the user with a list of permissions the originating application is attempting to gain on the user’s account.

If the user confirms they want to allow this access, they will be redirected back to the original site, along with an access token. This access token is then saved by the originating application.

Like a developer API key, this access token will be included on requests by the application to prove that the user has granted access and enable access to the appropriate content for that user. When a user returns to the application, the token will be retrieved and they will not have to re-authenticate.

## OAuth 2

Since OAuth evolved out of Twitter, there were important use cases not originally considered as part of the specification. Eventually, this led to the creation of a new version of the specification, called OAuth 2.

Among other improvements, OAuth 2 allows for different authentication flows depending on the specific application requesting access and the level of access being requested.

OAuth 2 is still an open standard, so each API will have its own flow based on its particular implementation. Below, we’ll discuss a few of the common OAuth 2 flows and how they are used.

## Client Credentials Grant

Sometimes an application will not need access to user information but may implement the added security and consistency of the OAuth 2 specification. This type of grant is used to access application-level data (similar to the developer API key above) and the end user does not participate in this flow.

Instead of an API key, a client ID and a client secret (strings provided to the application when it was authorized to use the API) are exchanged for an access token (and sometimes a refresh token). We will discuss refresh tokens in more depth later.

This flow is similar to our first example, where an email and password were exchanged for an authentication token.

**It is essential to ensure the client secret does not become public information, just like a password. As a result, developers should be careful not to accidentally commit this information to a public git repository. Additionally, to ensure integrity of the secret key, it should not be exposed on the client-side and all requests containing it should be sent server-side.**

Similar to the previously-mentioned keys, the returned access token is included on requests to identify the client making the requests and is subject to API restrictions.

This access token is often short-lived, expiring frequently. Upon expiration, a new access token can be obtained by re-sending the client credentials or, preferably, a refresh token.

Refresh tokens are an important feature of the OAuth 2 updates, encouraging access tokens to expire often and, as a result, be continuously changed (in the original OAuth specification, access tokens could last for time periods in the range of years). When a refresh token is used to generate a new access token, it typically expires any previous access tokens.

## Authorization Code Grant

This flow is one of the most common implementations of OAuth and will look familiar if you’ve ever signed into a web application with Google or Facebook. It is similar to the OAuth flow described earlier with an added step linking the requesting application to the authentication.

A user is redirected to the authenticating site, verifies the application requesting access and permissions, and is redirected back to the referring site with an authorization code.

The requesting application then takes this code and submits it to the authenticating API, along with the application’s client ID and client secret to receive an access token and a refresh token. This access token and refresh token are then used in the same manner as the previous flow.

To avoid exposing the client ID and secret, this step of the flow should be done on the server side of the requesting application.

Since tokens are tied both to users and requesting applications, the API has a great deal of control over limiting access based on user behavior, application behavior, or both.

## Implicit Grant

The previous two methods cause the client secret key to be exposed, so they need to be handled server-side. Some applications may need to access an OAuth API but don't have the necessary server-side capabilities to keep this information secure.

The Implicit Grant OAuth flow was designed for this very use case. This flow prompts the user through similar authorization steps as the Authorization Code flow, but does not involve the exchange of the client secret.

The result of this interaction is an access token, and typically no refresh token. The access token is then used by application to make additional requests to the service, but is not sent to the server side of the requesting application.

This flow allows applications to use OAuth APIs without fear of potentially exposing long-term access to a user or application's information.

## Conclusion

OAuth provides powerful access to a diverse set of sites and information. By using it correctly, you can reduce sign-up friction and enrich user experience in your applications.