‍[OAuth](https://www.oauth.com/) (pronounced "oh-auth”) is an authorization framework that allows an application to request “[secure delegated access](https://www.upguard.com/blog/access-control)” to third-party systems on behalf of the apps’ users or the “resource owner.”

Simply put, with OAuth, users can grant websites and applications access to their information on other websites without providing [important credentials](https://www.upguard.com/blog/third-party-credentials-vendor-risk) like passwords.

OAuth stands for "Open Authorization”. It’s an open standard that anyone can implement, and it’s usually used by devices, APIs (Application Programming Interface), web servers, and applications.

OAuth works as a middleman on behalf of the end-user and grants the third-party application access tokens for authorization. For example, OAuth is used when a web application requests access to a device’s camera or microphone.

The OAuth framework allows users to give another service or web application an access authentication token for authorization to protected resources. The process is validated via an authentication provider that the user has authenticated with.

OAuth Workflow Example

There are countless use cases that highlight the concept of OAuth. However, one of the best examples of OAuth is by an [early implementer](https://oauth.net/about/introduction/), which describes OAuth as a car’s valet key.

Unlike a regular key that allows the holder unlimited access, a valet uses this valet key only to take the car for a temporary spin without having access to the glove compartment or trunk.

OAuth enables an end user's account information to be used by third-party services like Amazon, Facebook, Google, Instagram, LinkedIn, Netflix, Paypal, Microsoft, and Twitter.

These companies work with many third-party applications and frequently permit users to share their credentials within apps like news websites, blogging platforms, online video games, etc.

When logging into another app on Facebook, a user may click the Facebook login button. Then, Facebook automatically authenticates them, and the website grants them access with permission from Facebook.

Moreover, OAuth is also used for email authentication, like Gmail, in which a web service gains access to protected resources within another device, allowing the user to access two different services with one login.

History of OAuth

First introduced in 2007, OAuth 1.0 was developed from the improvement process of the OpenID standard as an authorization procedure for Twitter’s API. Developers were figuring out how to secure access to multiple systems instead of one but on behalf of a sole client, and they stumbled upon the concept of OAuth.

Strongly supported by Twitter and Google, among other giants, the IETF OAuth Working Group introduced the blueprint and first draft of the OAuth 2.0, which was released as an open standard in 2010 as [RFC 5849](https://tools.ietf.org/html/rfc5849).

In 2012, after significant changes, improvements, and revisions, the final version of 2.0 OAuth was released as [RFC 6749](https://tools.ietf.org/html/rfc6749). Tech giants adopted the new OAuth 2.0, and everyone else followed suit.

While OAuth 2.0 was heavily criticized mainly for [security reasons](https://www.upguard.com/blog/cybersecurity-important) and complexity, it became a very popular online authorization standard. Today, it’s the most widely used version, replacing the prior OAuth 1.0 version.

Much like the first version, OAuth 2.0 allows users to grant third-party applications access to their credentials without giving away credentials.

The updated characteristics of 2.0 include a new authorization code flow, which works best for mobile applications, simplified signatures, and access tokens with a short time span but long-lasting authorization power.

What’s the Difference Between Oauth 1.0 and Oauth 2.0?

OAuth 2.0 is a completely rewritten framework of OAuth 1.0, and the 1.0 version is regarded as a protocol that uses different terminology and terms.

The main difference in function is how both versions categorize duties and end-user experience.

OAuth 1.0 has a consumer, service provider, and user, and it doesn’t explicitly separate the roles of the resource server and authorization server.

On the other hand, OAuth 2.0 has a client, authorization server, resource server, and resource owner. That’s why these versions don’t have backward compatibility.

OAuth 2.0 offers specific authorization flows for web applications, desktop applications, mobile phones, [smart devices](https://www.upguard.com/blog/internet-of-things-iot), and non-browser-based applications such as API-based services.

How OAuth 2.0 Works

Before learning how OAuth 2.0 works, it’s important to know about OAuth 2.0 roles.

The main four roles in an OAuth 2.0 authorization framework are:

1. Resource Owner

The user, or the owner of the [protected resources](https://www.upguard.com/blog/personally-identifiable-information-pii) (e.g., credentials, account info, contact lists, [sensitive data](https://www.upguard.com/blog/sensitive-data)), and this entity has the power to grant an application access to their protected resources or credentials.

2. Client

The application that sends an access request to the resource owner’s protected resources. To attain access, the client attempts to obtain an access token from the authorization server. When requesting an access token, a client has to acquire its own credentials, a client ID, and a client secret from the authorization server for identification and authentication. The client can be a web application, mobile app, TV app, or other application.

3. Resource Server

Also called the API server in OAuth 2.0, the Resource Server is the external server that a user integrates with. A resource server safeguards the user’s credentials and resources. Examples of resource servers include Google, DropBox, and Facebook. A client may request access from the resource server. The resource server is in charge of handling authenticated requests once the app receives an access token.

4. Authorization Server

The Authoritzation server accepts credentials from the resource owner, checks to see if they’re authorized to access the resource. After successful authentication, an authorization server gives away the tokens via the resource owner’s consent. The authorization server has an authorization endpoint that manages the resource owner’s consent user and a token endpoint that deals with M2M (machine-to-machine communication).

How the OAuth Flow Works

A client initiates access requests to the resources via an authorization code flow for requesting, obtaining, and using an access token. There are many OAuth flow types, such as client credentials flow, resource owner password flow, implicit flow with form post, hybrid flow, device authorization flow, and authorization code flow with PKCE.

1. In order to access the protected resources that are owned by the resource owner (the user), the client (application) sends an authorization request to the authorization server (API) to request authorization by conducting the user to the authorized endpoint. The client supplies the client ID token as an instrument for identification

2. The resource owner validates the application’s access request by authenticating and authorizing it. The client provides an endpoint URI, or a redirect URI, to which it sends the authorization grant (authorization code), and it also presents the scopes that the resource owners authorize. Then, the authorized endpoint gives the client an OAuth 2.0 protocol authorization grant.

3. The client requests an access token (in URL form) from the authorization server by exchanging the token for an authorization grant. The grant, as well as the authentication of the grant’s identity, is returned to the token endpoint.

4. If the authorization grant is valid and the client identity is authenticated, the client is given an access token either by the provider of the authentication or the authorization server (for example, Twitter’s Authorization Server).

5. With this authentication access token, the client is allowed to request the credentials from the resource server.

6. Once the access token is validated, the resource server (API) gives the credentials back to the client.

Access Tokens, Authorization Codes, and Grant Types in OAuth 2.0

Access tokens are files of a long string of characters that clients use as a form of replacement for credentials for accessing resources hosted by the resource server.

These tokens are generated by OAuth in a JWT format (JSON Web Token, not to be confused with [bearer tokens](https://oauth.net/2/bearer-tokens/)), which is an authentication standard that uses signatures for verification. The access token represents permissions known as scopes. More on that later.

OAuth 2.0’s access tokens are valid for a short period of time. Depending on the permissions granted, they can last from five minutes to a week.

Access tokens should have been called “temporary slips” for clarity, but as time went on, “access tokens” became widely accepted.

The use of access tokens in the authorization process allows users to give an application permission to their data within another third-party application without providing access to their password.

With the resource owner’s approval, OAuth allows the issuing of access tokens to third-party clients via authorization servers.

However, refresh tokens can be utilized in order to get a new access token without repeating the authorization process.

These tokens contain information about:

* the authentication session;
* the user identifier;
* an identifier for the identity provider who issued the token;
* length of time they'll be valid;
* the length of time that has passed since the authorization process began;
* and an identifier for the client for which the token was created.

OAuth 2.0’s security is enforced via [TLS/SSL](https://www.upguard.com/blog/ssl-certificate) [Hypertext Transfer Protocol (HTTPS)](https://www.upguard.com/blog/what-is-https) for all communications. The OAuth tokens are typically sent at layer 7 of the OSI model [using HTTPS](https://www.upguard.com/blog/what-is-https) to ensure neither the client nor server need to [generate a signature](https://www.upguard.com/blog/ids-vs-ips) to [encrypt communication](https://www.upguard.com/blog/encryption) between the two.

Authorization Codes and Grant Types

Not to be confused with an access token, an authorization grant gives the client permission to retrieve an access token on behalf of the end user.

There are a several grant types with different characteristics that are defined by the OAuth 2.0 framework:

* [**Authorization Code**](https://auth0.com/docs/api-auth/tutorials/authorization-code-grant): A single-use code that the authorization server returns to the client to be exchanged for an access token. It’s usually the most common method for standard web apps and mobile/native apps in which the exchange safely occurs on the server side. The authentication is processed through the client ID instead of the client secret because of safety concerns.
* [**Authorization Code Grant with Proof Key for Code Exchange (PKCE)**](https://auth0.com/docs/flows/concepts/auth-code-pkce): A more secure authorization flow than the standard authorization code grant, involving extra steps for added security, especially for native apps and single-page applications.
* [**Implicit Grant**](https://auth0.com/docs/api-auth/tutorials/implicit-grant): Astraightforward authentication flow in which the access token is given back to the client by the authorization server. It can grant an access token to integrations that are unable to [keep sensitive data](https://www.upguard.com/blog/sensitive-data). With an implicit flow protocol, the access token is returned as a response to a form post that has its potential security issues concerning token leakage or as a query parameter in the callback URI.
* [**Resource Owner Credentials Grant Type**](https://auth0.com/docs/api-auth/tutorials/password-grant): Prompts a trusted client to request the resource owners’ credentials, which are moved to the authorization server. Theis grant type doesn’t redirect to the authorization server, making it viable in use cases in which redirecting is impractical.
* [**Client Credentials Grant**](https://auth0.com/docs/api-auth/tutorials/client-credentials): Used by clients to receive an access token outside of the context of a user, most commonly for automated processes. Here, the application is validated via client secret and client ID.
* [**Refresh Token Grant**](https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/)**:**If an access token expires, the refresh token type allows clients to exchange a refresh token for an access token. Unlike access tokens which, for security reasons, have a limited time to be used, refresh tokens last longer during authentication.
* [**Device Authorization Flow**](https://auth0.com/docs/flows/concepts/device-auth): While not a grant, a device authorization flow allows a device to prompt the user to set up a link on a different device and authorize the app instead of authenticating the user directly. This flow enhances user experience for input-constrained devices, like TVs and smartphones.

OAuth 2.0 Scopes

In the OAuth context, a scope is a mechanism that grants a web app limited access to a user’s data. Scopes are used as a way for apps to request more limited scopes of privileges that are allowed on the user’s behalf.

For example, there are apps that use OAuth to identify the user and only require a user’s ID. On the other hand, there are apps that request a user's [more detailed information, such as birth date and contact information.](https://www.upguard.com/blog/personally-identifiable-information-pii)

When an app shows what type of access they’re requesting, the user can then decide if they want to authorize the app or not.

Depending on the resource server, scopes can be an important concept in OAuth 2.0. Scopes are also crucial when used to specify the reason why resource access may be granted.

An OAuth 2.0 scope has a set of different permissions that correspond to one single type of function.

By using different scopes, a user can connect two independent clients at the same time, for example, to a Twitter account. The user can authorize one scope for sending tweets and communications and another for scrolling the timeline.

What's the Difference Between OAuth and OpenID?

OpenID is used for federated identity authentication, which lets third-party apps authenticate users with existing accounts. On the other hand, OAuth is used for authorization and privacy purposes, allowing users to not share their passwords with third-party applications.

While both are [HTTPS-based](https://www.upguard.com/blog/what-is-https) and can be utilized for similar jobs, they’re not interchangeable in use. Authentication deals with verifying identity, while authorization is giving access.

When a client uses OAuth, a server gives an access token to a third party to access a protected resource. The source validates the token, but the token owner’s identity is not verified.

A token issued by a resource server is much like a movie ticket in which there’s no identifying info about the person, but it’s a permission slip that grants entry. This also means that the token can be used by a malicious actor or machine.

Without revealing a person’s credentials to the application, OAuth 2.0 permits the service user to allow a third-party app access to their data hosted with their service.

OpenID Connect

OpenID Connect is an open standard that organizations use to authenticate users and permit third-party applications to obtain a user’s identity being managed by a service.

Based on OpenID Connect and OAuth 2.0 for a more secure login, this authentication option is used by Google Plus Sign-In for logging into YouTube or Facebook.

Dubbed as “[SAML](https://www.upguard.com/blog/what-is-saml) with curly brackets,” OpenID Connect extends OAuth 2.0 for authentication scenarios.

Identity Providers (IdPs) use OpenID to allow users to sign in to the IdP and then access other websites and apps without having to log in or share their sign-in information. This makes it easier for developers to authenticate their users across websites and apps without having to own and manage their passwords.

Identity Providers and Server Providers

Examples of identity providers are Microsoft Active Directory and Azure, while service providers are CRM solutions.

If a user wants access to an application, identity providers serve to authenticate the user's credentials and give the service provider valid authorization.

If a user wants authentication, service providers usually request authorization from their designated identity provider.

Since users generally authenticate with individual services and an app with one-to-one mapping with an IdP, OAuth tends to be more application-centric.

Typically, when a client registers via OAuth 2.0, it’s a one-time procedure in which the registration remains valid for a period of time, or the OAuth client revokes the registration.

OAuth isn’t purposed to prove credentials to an application that the person operating is really them. However, you still need to authenticate a fraction of your identity in order for OAuth to give third-party systems “secure delegated access.”

OAuth vs SAML

The difference between [SAML (Security Assertion Markup Language)](https://www.upguard.com/blog/what-is-saml) and OAuth is that the former is a federated, open-standard authentication protocol, while the latter is a standardized authorization system for granting limited access to online services.

SAML is focused on enterprise security and [single sign-on (SSO)](https://help.upguard.com/en/articles/3797584-what-sso-solutions-does-upguard-support) instances. Compared to OAuth, it's more user-centric, and it allows users to log into a server with a password.

Additionally, SAML implements a safe way of completing user authentication and authorization between an identity provider like Microsoft Active Directory or Azure and a service provider like Salesforce and other CRM solutions.

SAML authorization has the ability to command the service provider what level of access to grant the authenticated user.

For exchanging messages, SAML uses XML and session cookies, while OAuth uses [JSON](https://theserverside.techtarget.com/definition/JSON-Javascript-Object-Notation) and API calls. Depending on the time zone and location, the use of session cookies is good for authorization and access during the day, but it’s not convenient for IoT devices, mobile apps, or other devices.