**12/28/21, 10/12/22**

[**https://carto.com/developers/auth-api/guides/how-to-send-API-Keys/**](https://carto.com/developers/auth-api/guides/how-to-send-API-Keys/)

[**https://stormpath.com/blog/token-authentication-scalable-user-mgmt**](https://stormpath.com/blog/token-authentication-scalable-user-mgmt)

[**https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-production**](https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-production)

**https://auth0.com/docs/flows/authorization-code-flow**

[**https://auth0.com/docs/tokens/json-web-tokens**](https://auth0.com/docs/tokens/json-web-tokens)

[**https://auth0.com/docs/tokens/refresh-tokens/configure-refresh-token-expiration**](https://auth0.com/docs/tokens/refresh-tokens/configure-refresh-token-expiration)

[**https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-production#use-access-tokens**](https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-production#use-access-tokens)

[**https://auth0.com/docs/get-started**](https://auth0.com/docs/get-started)

[**https://auth0.com/docs/universal-login/universal-vs-embedded-login**](https://auth0.com/docs/universal-login/universal-vs-embedded-login)

[**https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-testing#get-access-tokens-manually**](https://auth0.com/docs/tokens/management-api-access-tokens/get-management-api-access-tokens-for-testing#get-access-tokens-manually)

[**https://www.oauth.com/**](https://www.oauth.com/)

[**https://smartbear.com/ppc/ready-api/soapui/?utm\_source=aw&utm\_medium=ppcg&utm\_term=%2Btesting%20%2Bapis&utm\_content=469733882335&utm\_campaign=342047661&gclid=Cj0KCQiAlsv\_BRDtARIsAHMGVSYO29HVmexPeBdgO8MIOJhwspR7IUwzy0jQ8Px-ZGll5A9un\_Y3vg0aAijLEALw\_wcB&gclsrc=aw.ds**](https://smartbear.com/ppc/ready-api/soapui/?utm_source=aw&utm_medium=ppcg&utm_term=%2Btesting%20%2Bapis&utm_content=469733882335&utm_campaign=342047661&gclid=Cj0KCQiAlsv_BRDtARIsAHMGVSYO29HVmexPeBdgO8MIOJhwspR7IUwzy0jQ8Px-ZGll5A9un_Y3vg0aAijLEALw_wcB&gclsrc=aw.ds)

Get Management API Access Tokens for Production

To make scheduled frequent calls for a production environment, you have **to build a process at your backend that will provide you with a token automatically** (and thus simulate a non-expiring token).

Prerequisites

* [Create and Authorize a Machine-to-Machine Application](https://auth0.com/docs/tokens/management-api-access-tokens/create-and-authorize-a-machine-to-machine-application).

**Get access tokens**

To ask Auth0 for a Management API v2 token, perform a POST operation to the **https:**//YOUR\_DOMAIN/**oauth/token** endpoint, using the credentials of the Machine-to-Machine Application you created in the prerequisite step.

The payload should be in the following format:

curl --request POST \

--url 'https://YOUR\_DOMAIN/oauth/token' \

--header 'content-type: application/x-www-form-urlencoded' \

**--data grant\_type=client\_credentials** \

--data '**client\_id**=YOUR\_CLIENT\_ID' \

--data **client\_secret**=YOUR\_CLIENT\_SECRET \

--data 'audience=https://YOUR\_DOMAIN/api/v2/'

# **python to make HTTPS requests**

import **http.client**

**conn** = http.client.**HTTPSConnection**("")

**payload** = "**grant\_type**=client\_credentials**&client\_id**=%24%7Baccount.clientId%7D**&client\_secret**=YOUR\_CLIENT\_SECRET&audience=https%3A%2F%2F%24%7Baccount.namespace%7D%2Fapi%2Fv2%2F"

**headers** = { 'content-type': "application/x-www-form-urlencoded" }

**conn.request**("POST", "/YOUR\_DOMAIN/oauth/token", payload, headers)

res = conn.getresponse()

data = res.read()

print(data.decode("utf-8"))

#-----------------------

The **response** will contain a [**signed JWT**](https://auth0.com/docs/tokens/json-web-tokens), an expiration time, the **scopes** granted, and the token type.

{

"**access\_token**": "eyJ...Ggg",

"**expires\_in**": 86400,

"**scope**": "read:clients create:clients read:client\_keys",

"**token\_type**": "Bearer"

}

From the above, we can see that our Access Token will expire in 24 hours (86400 seconds), it has been authorized to read and create applications, and it is **a**[**Bearer Access Token**](https://tools.ietf.org/html/rfc6750).

# Revoke Tokens

Once issued, access tokens and ID tokens **cannot be revoked in the same way as cookies** with session IDs for server-side sessions.

As a result, **tokens should be issued for relatively short periods**, and then [**refreshed**](https://auth0.com/docs/tokens/refresh-tokens)periodically if the user remains active.

**12/27/21**

Some APIs use API keys **for authorization**. An API key is a token that a client provides when making API calls. The key can be sent in the query string:

**API keys** are public, by intent. They are an **authorization mechanism**, **not an authentication mechanism**. It does not matter how they are *generated* but it matters how they are *handled*. In other words: "anyone with this key can enter".

So, **you use API keys when you want to authorize and do not need to authenticate**.

A typical API key for a REST-ful application usually happens to be significantly less secure than the access control provided by **an OAuth JWT** (JSON Web Token)

Due to the fact that the API key string is being passed as an HTTP **GET query** parameter, it is much easier for intermediate web servers (including proxies), and browsers with client-side scripting languages such as JavaScript or ActionScript to gain read and/or write access to the API key.

1. GET /something**?api\_key=abcdef12345**
2. as a request header

GET /something HTTP/1.1 2.

X-API-Key: abcde

**OAuth access tokens** on the other hand, are generated on a **per-session basis**. Being granted an access token by a secure authentication provider will not occur, until the provider has received proof that the requesting user is entitled to requested privileges; such proof might be established through knowledge of credentials (i.e. a corresponding username and password pair.) Other times, **access control might be more restrictive and access tokens are only provided for a small sub-set of privileges** within a particular app/site/API sub-component, area of operation, control sphere, etc. The permissions ultimately granted to the end user can be as fine-grained as the system administrator wishes. Note that **access tokens are programmed to expire** after a set amount of time and are capable of providing discretionary access control between various users/groups, privileges/capabilities, etc.

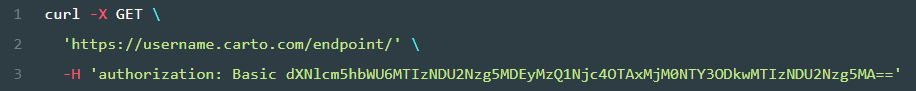
Access tokens are often transferred outside of the URL in the HTTP request header's Authorization field, for example. Sometimes, custom authentication framework implementations will cause the token to be transmitted within a cookie that has the **HttpOnly**, **Secure** and **SameSite** flags enabled--or as a custom HTTP request header such as **X-Auth-Token** as publicly documented for Oracle's Cloud Storage SaaS: [Oracle's Cloud Storage Service API](https://docs.oracle.com/en/cloud/iaas/storage-cloud/cssto/authenticating-access-when-using-rest-api.html):

It is extremely rare for HTTP request headers and cookie values to be logged by web browser/server software; they're also more difficult to access programatically due to CORS (Cross Origin Resource Sharing.) In comparison, the API keys passed as HTTP GET parameters can be extracted with client-side JavaScript from the DOM (Document Object Model).

For these reasons, the complexity required to obtain access tokens from an authentication framework such as OAuth is much higher than what is needed to log the usage of an API key. Furthermore, the robustness of authentication and authorization frameworks allows the access token to be encapsulated within the HTTP protocol in ways that it is rather difficult to view or tamper with the token.

* [**https://carto.com/developers/auth-api/guides/how-to-send-API-Keys/**](https://carto.com/developers/auth-api/guides/how-to-send-API-Keys/)

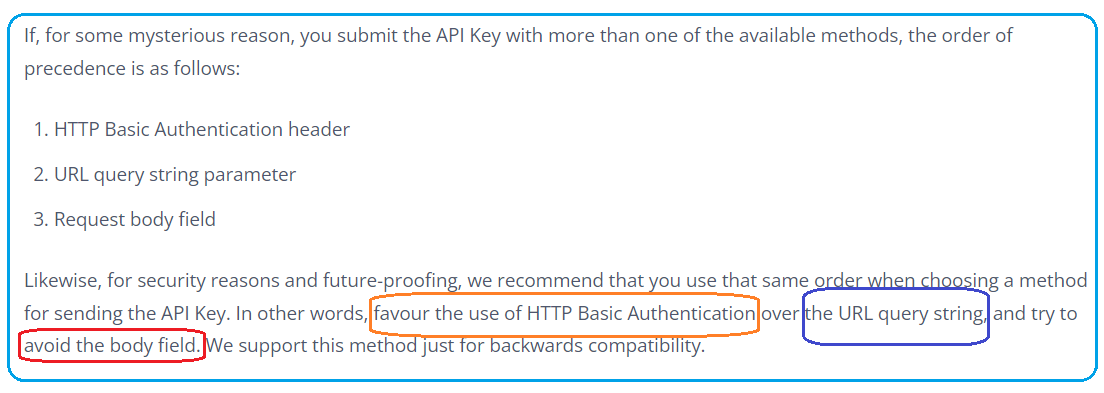
For requests to CARTO’s APIs, **take the API Key as the password**, and the username as the user who issued that API Key.

****

### **Query string/Request body parameter**

Alternatively, you can use an URL query string parameter or a field in the request body. In both cases, the name of the parameter is api\_key.



****

# Authentication vs Authorization

* [**https://auth0.com/intro-to-iam/authentication-vs-authorization/**](https://auth0.com/intro-to-iam/authentication-vs-authorization/)

## What is **Authentication**?

Authentication is the process of **confirming the identity** of a user or a device (i.e., an entity). During the authentication process, an entity usually relies on some proof to authenticate itself, i.e. an **authentication factor**. For example, if you go to the bank and try to withdraw money from your account, the clerk might ask you for a user identity document to check who you are. Along the same lines, if you buy a flight ticket, you might need to use a passport to prove you are the person entitled to use that ticket before hopping on the plane. Both examples illustrate real-life situations where authentication processes take place to confirm your identity (authenticated user).

In a digital transaction, for example, when you try to access your Facebook profile or your company webmail client, a similar process happens. In these situations, **instead of presenting an ID, a passport, or similar**, you usually **prove your identity by showing the system that you know something (like a username and password)** or that you own a device (like a mobile phone so you can receive an SMS with a code). After presenting this knowledge or proving that you control a particular device, **the targeted system recognizes your identity and lets you access it.** In this scenario, the authenticated user uses authentication factors to prove their identity. These factors can be single, two-factor authentication, or multi-factor authentication.

## What is **Authorization**?

In contrast to authentication, authorization refers to t**he process of verifying what resources entities (users or devices) can access**, or what actions they can perform, i.e., their access rights.

For a concrete example, imagine a situation where you **buy a ticket for a show**. In this case, more often than not, the establishment will not be interested in your identity (i.e., who you are). What they care about is whether you are authorized or not to attend the show. To prove that you have the right to be there, instead of using an ID or a passport, you would use a ticket.

Often, **the ticket that authorizes you to attend the show does not contain any information about your identity**. However, even if it includes information about your identity, it is not what is verified in the authorization process.

In internet-based software applications, a common approach is to use artifacts called tokens to handle authorization. Typically**, once a user is signed in**, applications start caring about what they can do. In this scenario, **this leads to the creation of a token that carries authorization details based on the user identity**. The system uses this authorization token to make authorization decisions; this grants or prevents a request to access resources.

Although the sections above can shed light on what authentication and authorization stand for, the definition and usage of these terms may frequently overlap (which may be the root cause of the overall confusion about them). For example, in the bank scenario, the user identity presented to the clerk is also used to authorize access to the funds in your account.

In a similar scenario, a company that requires badges to control access between rooms **uses these badges to both authenticate the person (name and picture) and to authorize access**.

So you see, authentication and authorization are topics that seem to be interchangeable in certain scenarios; it is this that causes confusion.

However, the important point is that authentication leads to authorization, but authorization does not lead to authentication.

* [**https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/**](https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/)

This post will explore the concept of refresh tokens as defined by [OAuth 2.0](https://auth0.com/docs/protocols/protocol-oauth2). We will learn how they compare to other token types and how they let us balance security, usability, and privacy.

## **What Is A Token**?

**Tokens** are **pieces of data** that carry just enough information to facilitate the process of determining a user's **identity** or **authorizing a user to perform an action**. All in all, tokens are artifacts that allow application systems to perform the authorization and authentication process.

Common identity frameworks and protocols use token-based strategies to secure access to applications and resources. For example, we can use OAuth 2.0 for authorization and OIDC for authentication.

[**OAuth 2.0**](https://auth0.com/intro-to-iam/what-is-oauth-2/) is one of the most popular **authorization** frameworks out there. It is designed to allow an application to access resources hosted by other servers on behalf of a user. OAuth 2.0 uses **Access Tokens** and **Refresh Tokens**.

[**OpenID Connect** (OIDC)](https://auth0.com/intro-to-iam/what-is-openid-connect-oidc/) is an identity protocol that performs user **authentication**, user consent, and token issuance. OIDC uses **ID Tokens**.

Let's explore the **three token types** that we use with **OAuth 2.0** and OpenID Connect to fulfill the authentication and authorization processes of our application systems. In the process, we'll see the critical role that refresh tokens play in helping developers build applications that offer convenience without compromising security.

## Token Types

### **What's an ID token?**

As the name may suggest, an [**ID token**](https://auth0.com/docs/tokens/id-tokens) is an artifact that client applications can use to consume the identity of a user. For example, the ID token can contain information about the name, email, and profile picture of a user. As such, client applications can use the ID token to build a user profile to personalize the user experience.

An **authentication server** that conforms to the **OpenID Connect (OIDC) protocol** to implement the authentication process issues its clients an ID token whenever a user logs in. The consumers of ID tokens are mainly client applications such as Single-Page Applications (SPAs) and mobile applications. They are the intended audience.

### **What's an access token?**

**When a user logs in**, the **authorization server** issues an [**access token**](https://auth0.com/docs/tokens/access-tokens), which is an artifact that client applications can use to make secure calls to an API server. When a client application needs to access protected resources on a server on behalf of a user, the access token lets the client signal to the server that it has received authorization by the user to perform certain tasks or access certain resources.

**OAuth 2.0 doesn't define a format for access tokens**. At **Auth0**, for example, access tokens issued for the Management API and access tokens issued for any custom API that you have registered with Auth0 **follow the**[**JSON Web Token (JWT) standard**](https://jwt.io/). Their basic structure conforms to the typical JWT structure, and they contain standard JWT claims asserted about the token itself.



It's important to highlight that the **access token is a bearer token**. **Those who hold the token can use it**. The access token then acts as a credential artifact to access protected resources rather than an identification artifact. Malicious users could theoretically compromise a system and steal access tokens, which in turn they could use to access protected resources by presenting those tokens directly to the server.

As such, it's critical to have security strategies that minimize the risk of compromising access tokens. One mitigation method is to create access tokens that have a short lifespan: they are only valid for a short time defined in terms of hours or days.

**There are different ways that a client application can get a new access token for a user**. For example,

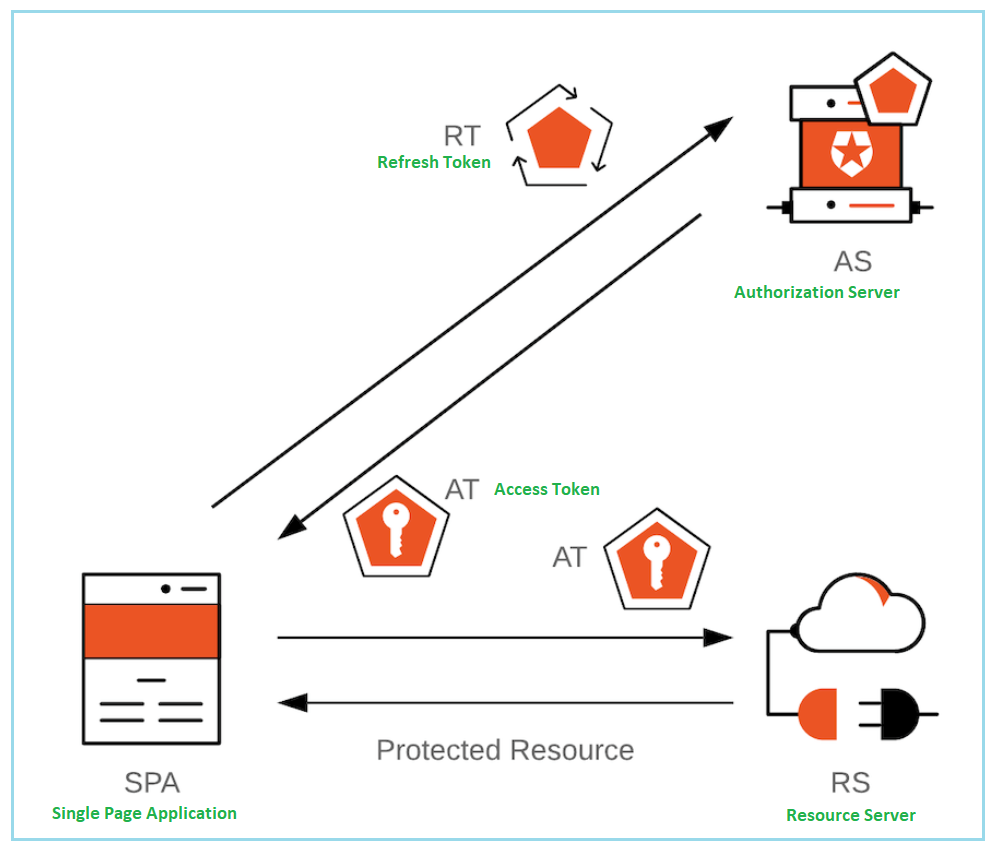
1. once an access token expires, the client application could **prompt the user to log in again** to get a new access token.
2. Alternatively, the authorization server could **issue a refresh token** to the client application that lets it replace an expired access token with a new one.

You can **see both ID tokens and access tokens in action** in any of our "Complete Guides to User Authentication" available for [React](https://auth0.com/blog/complete-guide-to-react-user-authentication/), [Angular](https://auth0.com/blog/complete-guide-to-angular-user-authentication/), [Vue](https://auth0.com/blog/complete-guide-to-vue-user-authentication/), and[Node.js](https://auth0.com/blog/complete-guide-to-nodejs-express-user-authentication/)!

## What Is a **Refresh Token**?

https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/

As mentioned, for security purposes, access tokens may be valid for a short amount of time. Once they expire, client applications can use a refresh token to "refresh" the access token. That is, a [**refresh token**](https://auth0.com/docs/tokens/refresh-tokens)**is a credential artifact that lets a client application get new access tokens without having to ask the user to log in again**.

The client application can get a new access token as long as the refresh token is valid and unexpired. Consequently, a refresh token that has a very long lifespan could theoretically give infinite power to the token bearer to get a new access token to access protected resources anytime. The bearer of the refresh token could be a legitimate user or a malicious user. As such, security companies, such as Auth0, create mechanisms to ensure that this powerful token is mainly held and used continuously by the intended parties.

## When to Use Refresh Tokens

It's important to keep in mind that the **OAuth 2.0 specification defines access tokens and refresh tokens.** So, if we were to discuss authorization strategies in terms of other identity protocols or frameworks, such as SAML, we would not have the concepts of access tokens or refresh tokens.

For those involved with web development, access token and refresh tokens are common talk because the web extensively uses token-based authorization and authentication through the OAuth 2.0 framework and the OpenID Connect protocol.

When combined, OAuth 2.0 and OIDC bring to life an array of authorization and authentication flows. Each flow has its own set of benefits and caveats that define the best scenarios and architecture where we should use access and refresh tokens.

* Is the client a **traditional web application** executing on the server? Use the [Authorization Code Flow](https://auth0.com/docs/flows#authorization-code-flow).
* Is the client a **Single-Page Application** (SPA)? Use [Authorization Code Flow with Proof Key for Code Exchange (PKCE)](https://auth0.com/docs/flows#authorization-code-flow-with-proof-key-for-code-exchange-pkce-).
* Is the client a Single-Page Application (SPA) that doesn't need an access token? Use the [Implicit Flow with Form Post](https://auth0.com/docs/flows#implicit-flow-with-form-post).
* Is the client the **resource owner**? You may use the [Client Credentials Flow](https://auth0.com/docs/flows#client-credentials-flow).
* Is the client absolutely trusted with user credentials? You may use the [Resource Owner Password Flow](https://auth0.com/docs/flows#resource-owner-password-flow).

If there's an app for that, there's also a flow for that!

Keep in mind that according to the spec, when using the Implicit Flow, the authorization server should not issue refresh tokens. The Implicit flow is often implemented in Single-Page Applications (SPAs), which run on the frontend layer of a system architecture. There's no easy way of keeping a refresh token secure in the frontend layer on its own.

Using the Authorization Code Flow with Proof Key for Code Exchange (PKCE) mitigates many risks inherent to the Implicit Flow. For example, when using the implicit grant type, the [access token is transmitted in the URI fragment](https://tools.ietf.org/html/rfc6749#section-10.3), which can expose it to unauthorized parties. You can learn more about these vulnerabilities by reading the ["Misuse of Access Token to Impersonate Resource Owner in Implicit Flow"](https://tools.ietf.org/html/rfc6749#section-10.16) section of the spec.

However, implementing PKCE in your applications still has no impact on how secure refresh tokens are.

However, you may not need refresh tokens.

There are scenarios where you can still get an access token without interrupting the user and without relying on the almighty power of the refresh token. Other examples to keep a session going can be cookies or [silent authentication](https://auth0.com/docs/authorization/configure-silent-authentication).

However, billions of people use SPAs every day. It is important to provide users with [a user experience that balances security and convenience well](https://auth0.com/blog/balance-user-experience-and-security-to-retain-customers/). Is there anything that we could do to let SPAs afford the convenience of refresh tokens in a less risky and more secure manner?

Absolutely!

[An identity platform that offers Refresh Token Rotation](https://auth0.com/blog/securing-single-page-applications-with-refresh-token-rotation/) makes it acceptable to use refresh tokens with Single-Page Applications. The [spec underlines](https://tools.ietf.org/html/rfc6749#section-10.4) that when you can not verify that a refresh token belongs to a client, such a SPA, we should not use them *unless we have Refresh Token Rotation in place*.

Let's learn more about this security strategy in the next section.

**Keeping Refresh Tokens Secure**

A short-lived access token helps improve the security of our applications, but it comes with a cost: when it expires, the user needs to log in again to get a new one. Frequent re-authentication can diminish the perceived user experience of your application. Even if you are doing so to protect their data, users may find your service frustrating or difficult to use.

A refresh token can help you balance security with usability. Since refresh tokens are typically longer-lived, you can use them to request new access tokens after the shorter-lived access tokens expire.

However, since **refresh tokens are also bearer tokens**, we need to have a strategy in place that limits or curtails their usage if they ever get leaked or become compromised. **All those who hold the refresh tokens have the power to get new access tokens whenever they want. "They" could be legitimate users or attackers.**

**At Auth0, we created a set of features that mitigate the risks associated with using refresh tokens by imposing safeguards and controls on their lifecycle**. Our identity platform offers refresh token rotation, which also comes with automatic reuse detection.

Let's dive deeper into this security technique.

**Refresh Token Rotation**

Until very recently, a robust strategy to help SPAs maintain the user's session was using the Authorization Code Flow with PKCE in conjunction with silent authentication. Refresh token rotation is a technique for getting new access tokens using refresh tokens that goes beyond [silent authentication](https://auth0.com/docs/authorization/configure-silent-authentication).

**Refresh token rotation** guarantees that every time an application exchanges a refresh token to **get a new access token, a new refresh token is also returned.** Therefore, you no longer have a long-lived refresh token that could provide illegitimate access to resources if it ever becomes compromised. The threat of illegitimate access is reduced as refresh tokens are continually exchanged and invalidated.

For example, with refresh token rotation enabled in the Auth0 Dashboard, every time your application exchanges a refresh token to get a new access token, the authorization server also returns a new refresh-access token pair. This safeguard helps your app mitigate [replay attacks](https://auth0.com/docs/security/prevent-common-cybersecurity-threats#replay-attacks) resulting from compromised tokens.

**Refresh Token Automatic Reuse Detection**

**Refresh tokens are bearer tokens. It's impossible for the authorization server to know who is legitimate or malicious when receiving a new access token request**. We could then treat all users as potentially malicious.

How could we handle a situation where there is a race condition between a legitimate user and a malicious one? For example:

**12/28/2021**

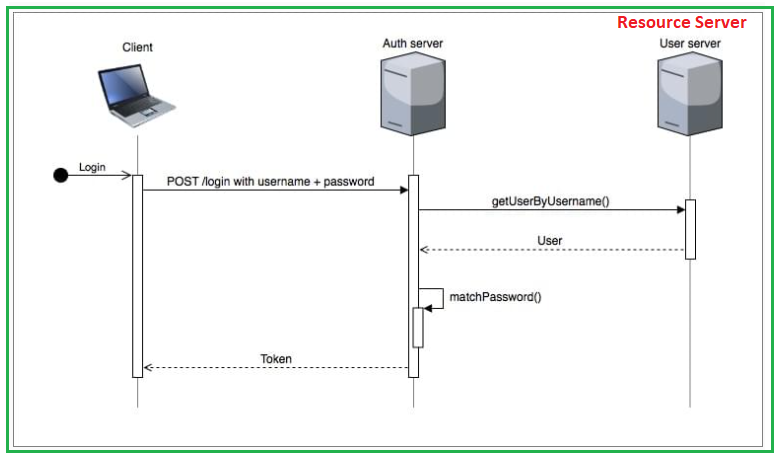
# Combining OAuth and JWT to gain performance improvements

**https://capgemini.github.io/architecture/combining-oauth-and-jwt-to-gain-performance-improvements/#main-content**

For many years **Simple Object Access Protocol** (SOAP) was the standard approach for communicating with remote services, often **via HTTP**. The landscape has changed significantly in recent years with the increase in the adoption of Representational State Transfer (REST) APIs. There are still a number of use cases that suit SOAP, for example where stateful operations are required.

One of the six guiding architectural principles of REST is **statelessness**. Every API request from the client to a server must contain all the necessary information necessary to serve the request. The server maintains neither state nor context. How do we then authorise access to protected REST APIs? Say hello to [OAuth](https://tools.ietf.org/html/rfc6749) and [JSON Web Token (JWT)](https://tools.ietf.org/html/rfc7519). OAuth and JWT are two of the most widely used token frameworks or standards for authorising access to REST APIs. In this blog post I consider how both OAuth and JWT can be combined to gain performance improvements.

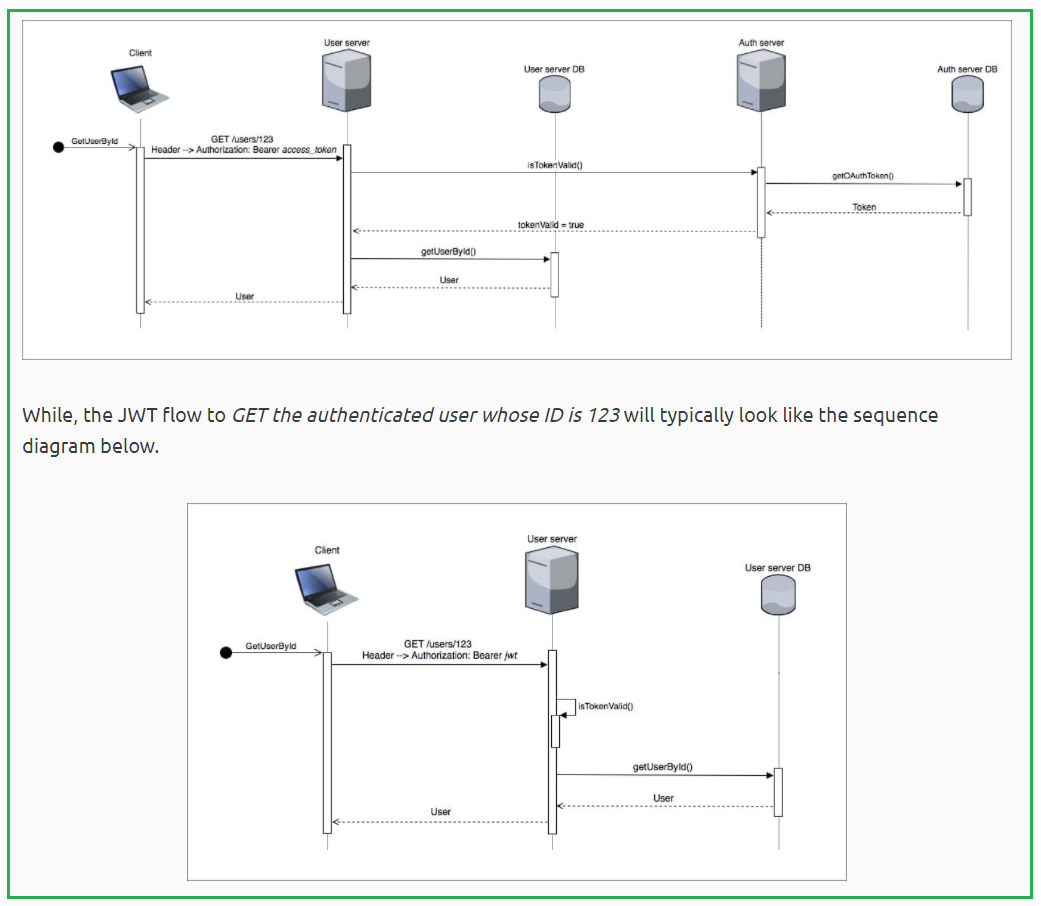
**OAuth enables an application to obtain limited access to an HTTP service**. While JWT is a compact, URL-safe means of representing claims to be transferred between two parties. OAuth has a number of grant types. So whenever I refer to OAuth in this blog post, I am referring to the [**OAuth 2.0** Resource Owner Password Credentials Grant type](https://tools.ietf.org/html/rfc6749#page-37). **A user is required to authenticate or login to obtain a token**. A typical authentication flow is shown in the sequence diagram below.



A user will **enter their username and password** via a client (which could be a mobile device or PC), and at the end of the authentication process the user will be supplied with a token.

**The client will then include the token with every subsequent API request to a resource server** (like the User server). To compare what the authorization flow for both OAuth and JWT will look like, let’s consider an example where we make an API request to *GET the authenticated user*.

The OAuth flow to *GET the authenticated user whose ID is 123* will typically look like the sequence diagram below.

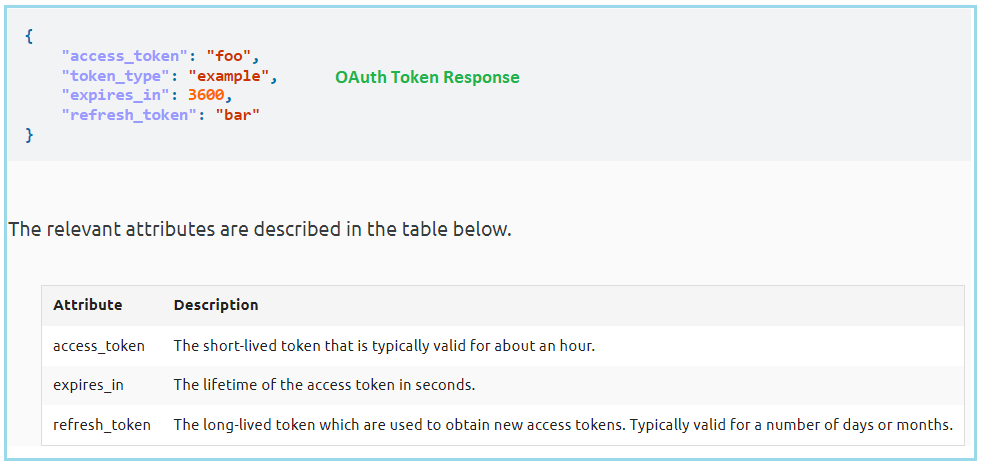


The JWT implementation is less chatty and more performant compared to OAuth. This is because **JWT enables a resource server to verify the token locally**. In its compact form, JWT consist of three parts: the **header**, **payload** and **signature**. The signature is the result of signing the base64Url encoded header and the base64Url encoded payload with a key. The **resource server uses the signature to verify that the token has not been tampered with**.

The JWT **payload contains the claims**. The claims are statements about an **entity** (typically, the user excluding private information of course) and additional metadata like the **expiry time**. JWT however has a drawback in that once it has been issued it will allow its holder to gain access to a resource server until the expiry time is lapsed. It looks like **we need a way to revoke JWTs**. So let’s go back to OAuth.

OAuth is chattier compared to JWT. This is because **OAuth** requires **the Auth server** to **verify the validity of the token** and the Auth server in turn relies on the information it has stored in a **database** to make this judgement.

**OAuth however does have an advantage over JWT in that tokens can be easily revoked**. This is particularly a good feature if instant access revocation is desired. The basic OAuth token response is shown below.

****

As shown in the example OAuth flow above, the client will include the access token in every API request to a resource server. The client will then make use of the refresh token to obtain a new access token. **How can we benefit from the inherent performance advantages associated with JWT and the limited access capability provided by OAuth**? By issuing an **OAuth token with JWT** in both access token and refresh tokens as depicted below.

****

The client will include the short-lived JWT for every call to the resource server, and will make use of the long-lived JWT to obtain a new access token. **The short-lived JWT is validated locally**.

We do however need to keep a record or **blacklist of the revoked refresh tokens till they expire**. **This blacklist will be checked only when the client wishes to refresh the OAuth token**. A new access token will not be granted if the refresh token is found in the blacklist. The blacklist should ideally contain refresh tokens associated with users who have logged out and users whose account have been disabled. **Although the access token is not immediately revoked, it is meant to be a short-lived token**.

Finally, there may be scenarios where this behavior is not desired but ultimately, it depends on the requirements of the system. This approach enables the resource server to validate the OAuth access token locally and only requires interaction with the Auth server when we need to get a new OAuth access token. It is this reduction in the interaction with the Auth server that gives us the performance improvements.

**Basically,**

**OAuth: needs authorization server and database to store access token and refresh token vs user; since tokens/user is stored in database, it can be revoked by adding to blacklist**

**JWT:**

**It can be verified locally in Resource Server. BUT it can’t be revoked**

**Combine Oauth framework with JWT token:**

**Access Token and Refresh token is just JWT token;**

**Access token is short-lived and verified locally by Resource Server; since it is short-lived, it doesn’t need to be revoked by Auth Server.**

**Refresh token: is long-lived and can be revoked by Auth server, by adding it to a blacklist maintained by Auth Server’s database until it expires. Once revoked refresh token expires, it can be removed from blacklist.**

**The OAuth 2.0 Authorization Framework**

[**https://datatracker.ietf.org/doc/html/rfc6749**](https://datatracker.ietf.org/doc/html/rfc6749)

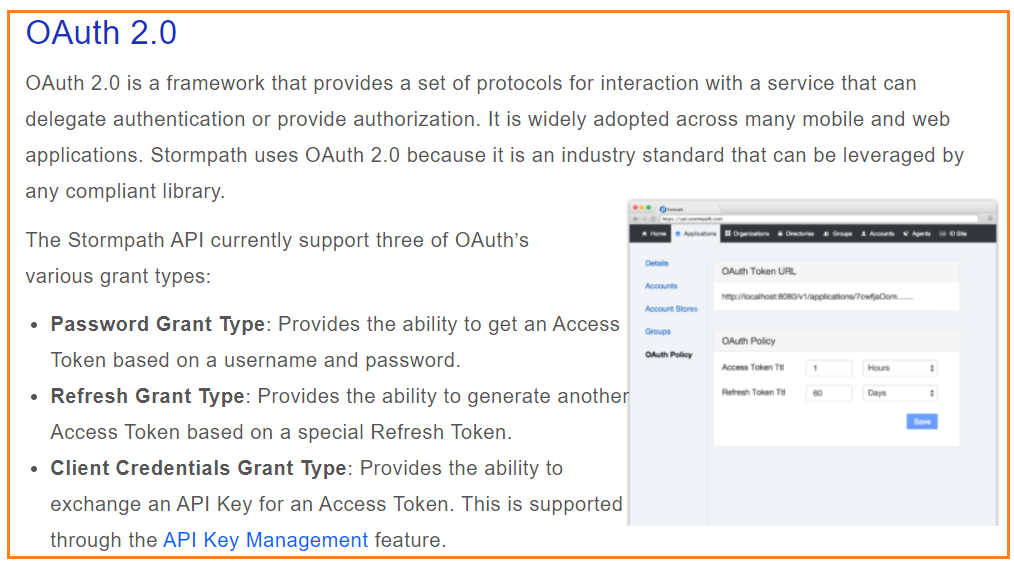
[**https://stormpath.com/blog/token-authentication-scalable-user-mgmt**](https://stormpath.com/blog/token-authentication-scalable-user-mgmt)

 A JWT is a compact, URL-safe, encryptable JSON object that is rapidly becoming the standard for token implementation across the web. A JWT looks like any other ugly string but is separated into three sections by periods.

The first section, or **header**, **describes** the **contents of the token**.

The second section, or **payload** (sometimes called “claims”), contains the identification data, authorization claims, and **expiration** time, as well as any custom data you choose to encode. (**identity of the user**)

The final section is the **signature**, a hash that cryptographically verifies the **validity of the token**.(make sure it is not tempered)



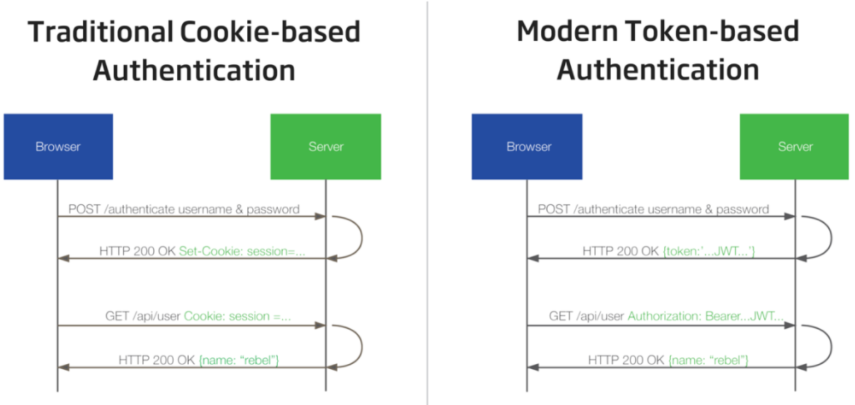
****

## **How Does Token Authentication Work?**

[Authentication](https://docs.stormpath.com/rest/product-guide/latest/auth_n.html) is the process by which an application confirms user identity. Applications have traditionally persisted identity through **session cookies**, relying on **session IDs** stored server-side. This forces developers to create session storage that is either unique to each server, or implemented as a totally separate session storage layer.

[Token authentication](https://docs.stormpath.com/rest/product-guide/latest/auth_n.html#how-token-based-authentication-works) is a more modern approach and is designed solve problems session IDs stored server-side can’t. Using tokens in place of session IDs can lower your server load, streamline permission management, and provide better tools for supporting a distributed or cloud-based infrastructure.

In this method, tokens are generated for your users after they present verifiable credentials. The initial authentication could be by  username/password credentials, API keys or even tokens from another service. (Stormpath’s [API Key Authentication Feature](http://docs.stormpath.com/guides/api-key-management/) is an example of this.)

****

s you can see in the diagram above, once the user’s credentials are exchanged for a token on the server, the client can use the token to validate each subsequent request. Once generated, the token is attached to the user via a browser cookie or [saved in local/session storage](https://stormpath.com/blog/where-to-store-your-jwts-cookies-vs-html5-web-storage/).

### **The Token Convention**

Let’s try an analogy: When you attend a conference or convention, you’re issued credentials specific to that event, often in the form of a lanyard. How did you score that lanyard? Well, you presented a valid, government-issued ID on the first day, they compared your ID, your face, and your registration. All matched, and the lanyard was yours. You’ve been authenticated. When you return to that convention the next day (a new session), or try to attend a gated sub-event (an authorization request), it’s your event credentials that are checked, not your actual ID.

In a session ID world, each gatekeeper would need a list of every single attendee and their registrations (permissions!), and would check your valid, government-issued ID against that list not just each day, but every time you entered a new area or session. That sounds exhausting, right?

In token land, gatekeepers authenticate your identity and authorize your access based on the custom event credential you’re wearing around your neck. No additional ID needed, no enormous list to check every individual against. The lanyard carries all the information!

## **Why use Token Authentication?**

### **1. Statelessness**

Here’s the crucial bit for **scalability**: Your server will need to generate a token, but it will never need to store said token anywhere. [All of the user metadata is encoded right into the token itself](https://stormpath.com/blog/token-auth-spa), so **any machine on your network** can validate any user. The server and client can pass the token back and forth forever and never store any user or session data. This is “statelessness,” and it’s the key to your application’s scalability.

**The only difficulties here is what if a token needs to be revoked before it expires. Oauth2.0 issues short – lived access token, while maintain a black-list in database only for revoked / invalid refresh token.**

### **2. Building a Mobile-Ready Backend**

Using tokens for [authentication in a mobile app](https://stormpath.com/blog/the-ultimate-guide-to-mobile-api-security)allow you to easily and securely control which mobile devices are accessing your API. Not only are they easier to use than cookies on iOS or Android, but they also allow your app to authenticate requests against multiple backends without extra effort on the part of your dev team.

### **3. Support for Multi-Server Platforms And Distributed Micro-Services**

Applications supported by multiple and distributed servers will reap the greatest benefits from tokens, simply because of their stateless nature.

Similar to our convention analogy, imagine a user logs into your app via Server A. Their first several requests are also sent to Server A as it is perhaps nearest to them geographically. That’s all well and good until a new request from that user triggers a shift to Server B, thanks to a load balancer or micro-service that’s hosted from Server B.

With traditional session-based authentication, Server B would have to be set up to talk to the same distributed session storage layer as Server A, or you would have to set up “sticky sessions” on your load balancer (thus severely reducing the value of your load balancer). Yikes!

With a token, Server B already has everything it needs to know to validate the user’s identity, no sticky sessions needed.

* I have written code for authenticating with **Salesforce**. I am using Grant\_type=Password. I am using some userName like "ABC" and password like "XYZ". I am making multiple request login with same credentials but i am getting same Access\_Token again and again. So my question

was-

1) Why i am getting same access token for multiple login request?

2) does Salesforce keeps user credential and Access\_Token mapping at his end and check if login request comes again for the same user it returns the previous Access\_Token?

Access Tokens have fixed Life-Time.

Until that period has not expires, access token remains the same. During that period if you request access token **you will receive the same string** but will **have reset expiration**. It would only happen if Location, Password etc are exactly same as previous request. Otherwise new token generated.

Sometimes Salesforce resets the expiration time for last expired token and return in

response.

**Salesforce keeps track of access tokens and manages them optimally to save their servers load**. Its all about server optimization with user security.

Firstly, we have to differentiate JWT and OAuth. Basically, JWT is a token format. OAuth is an

authorization protocol that can use JWT as a token. OAuth uses server-side and client-side

storage. If you want to do real logout you must go with OAuth2. Authentication with JWT token can not logout actually. Because you don't have an Authentication Server that keeps track of tokens. If you want to provide an API to 3rd party clients, you must use OAuth2 also. OAuth2 is very flexible. JWT implementation is very easy and does not take long to implement. If your application needs this sort of flexibility, you should go with OAuth2. But if you don't need this usecase scenario, implementing OAuth2 is a waste of time.

The JSON Web Token (JWT) tokens encodes all the data about the grant into the token itself. The most important advantage of this approach is that you do not need a backend store for token storage at all. **One disadvantage is that you can't easily revoke an access token,** so they normally are granted with short expiry and the revocation is handled at the refresh token. **Another disadvantage is that the tokens can get quite large** if you are storing a lot of user credential information in them. So if:

If I save on a database table the last tokens generated by the API, the "logout" would be done deleting them from the table of valid tokens

Then you would lose the most important advantage of using JWT and also, still have all those disadvantages, which seems unreasonable to me.

So, I understand that I shouldn't do that. The only solution that I'm thinking is make the JWT expiration time shorter, to 5 minutes, don't implement a "logout" operation and just let the tokens expire.

Is this the correct approach?

In my opinion, if you're planning to use JWT, YES! it's better to rely on the token expiration. For more details on this approach you can check [this question](https://stackoverflow.com/questions/26739167/jwt-json-web-token-automatic-prolongation-of-expiration) out.

Is “logout” useless on a REST API?

Regardless of the fact that you're using JWT and similar to any other decent questions on computer science, the answer would be It Depends. The most important advantage of Statelessness is that **your API would be more scalable**. If you choose this path, probably, every request on your API should be authenticated, since you may need to search a backend store for the given token or decode a JWT token. So, in this case you may have some performance cost on a single node but in a big picture, you would still have the scalability. I guess what i'm trying to say is, if you do not need that scalability, you're better off to choose a Stateful approach. Otherwise, pure REST principles is the way to go.

**Below are the steps to do revoke your JWT access token**:

1. When you do login, send 2 tokens (Access token, Refresh token) in response to client .  
   2) Access token will have less expiry time and Refresh will have long expiry time .  
   3) Client (Front end) will **store refresh token in his local storage** and **access token in cookies**, or used in authorization header, where token is not saved in cookies)  
   4) Client will use access token for calling apis. But when it expires, pick the refresh token from local storage and call auth server api to get the new token.  
   5) Your auth server will have an api exposed which will accept refresh token and checks for its validity and return a new access token.  
   6) Once refresh token is expired, User will be logged out.

When **encrypting**, you use **their public key** to write a message and they use **their private key** to read it.

When **signing**, you use **your private key** to write message's signature, and they use **your public key** to check if it's really yours.

There are two distinct but closely related problems in establishing a secure communication

1. **Encrypt data so that only authorized persons can decrypt and read it.**
2. **Verify the identity/authentication of sender.**

Both of these problems can be elegantly solved using public key cryptography.

**I. Encryption and decryption of data**

Alice wants to send a message to Bob which no one should be able to read.

* Alice encrypts the message with Bob's public key and sends it over.
* Bob receives the message and decrypts it using his private Key.

Note that if A wants to send a message to B, A needs to use the Public key of B (which is publicly available to anyone) and neither public nor private key of A comes into picture here.

So if you want to send a message to me you should know and use my public key which I provide to you and only I will be able to decrypt the message since I am the only one who has access to the corresponding private key.

II. **Verify the identity of sender (Authentication)**

Alice wants to send a message to Bob again. The problem of encrypting the data is solved using the above method.

But what if I am sitting between Alice and Bob, introducing myself as 'Alice' to Bob and sending my own message to Bob instead of forwarding the one sent by Alice. Even though I can not decrypt and read the original message sent by Alice(that requires access to Bob's private key) I am hijacking the entire conversation between them.

Is there a way Bob can confirm that the messages he is receiving are actually sent by Alice?

* Alice signs the message with her private key and sends it over. (In practice, what is signed is a hash of the message, e.g. SHA-256 or SHA-512.)
* Bob receives it and verifies it using Alice's public key. Since Alice's public key successfully verified the message, Bob can conclude that the message has been signed by Alice.