# Introduction

**Client**

**Resource Owner, Resource Server**

**Authorization server**

**Resource owner credentials**

**Client access token**

In the traditional client-server authentication model, **the client**

requests an access-restricted resource (protected resource) on the

server by authenticating with the server **using the resource owner's**

**credentials**. In order to provide third-party applications access to

restricted resources, **the resource owner shares its credentials with**

**the third party**. This creates several problems and limitations:

o Third-party applications are required to store the resource

owner's credentials for future use, typically a password in

clear-text.

o Servers are required to support password authentication, despite

the security weaknesses inherent in passwords.

o Third-party applications gain overly broad access to the resource

owner's protected resources, leaving resource owners without any

ability to restrict duration or access to a limited subset of

resources.

o **Resource owners cannot revoke access to an individual third party**

**without revoking access to all third parties, and must do so by**

**changing the third party's password.**

o Compromise of any third-party application results in compromise of

the end-user's password and all of the data protected by that

password.

**Oauth**

* **OAuth** addresses these issues by introducing an **authorization layer**

and separating the role of the client from that of the resource

owner. In OAuth, the ***client*** requests access to **resources** controlled

by the **resource owner** and hosted by **the resource server**, and is

issued **a different set of credentials** than those of the resource

owner.

* Instead of using the resource owner's credentials to access protected

resources, the client obtains an **access token -- a string denoting a**

**specific scope, lifetime, and other access attributes**. **Access tokens**

are issued to **third-party clients** by **an authorization server** **with the**

**approval of the resource owner**. The client uses the access token to

access the protected resources hosted by the resource server.

**?? How resource server authenticate Token?**

* For example, an **end-user** (resource owner) can grant a **printing**

**service (client)** access to her protected photos stored at a **photo-**

**sharing service (resource server)**, **without sharing her username and**

**password with the printing service**. Instead, she **authenticates**

directly with a server trusted by the photo-sharing service

**(authorization server**), which issues the printing service delegation-

specific credentials (**access token**).

* OAuth defines four roles:

**resource owner**

An entity capable of granting access to a protected resource.

When the resource owner is a person, it is referred to as an

end-user.

**resource server**

The server hosting the protected resources, capable of accepting

and responding to protected resource requests **using access tokens**.

**client**

An application **making protected resource requests** on behalf of the

resource owner and with its authorization. The term "client" does

not imply any particular implementation characteristics (e.g.,

whether the application executes on a server, a desktop, or other

devices).

**authorization server**

The server issuing **access tokens** to the **client** after successfully

**authenticating the resource owner** and obtaining authorization.

The interaction between the authorization server and resource server

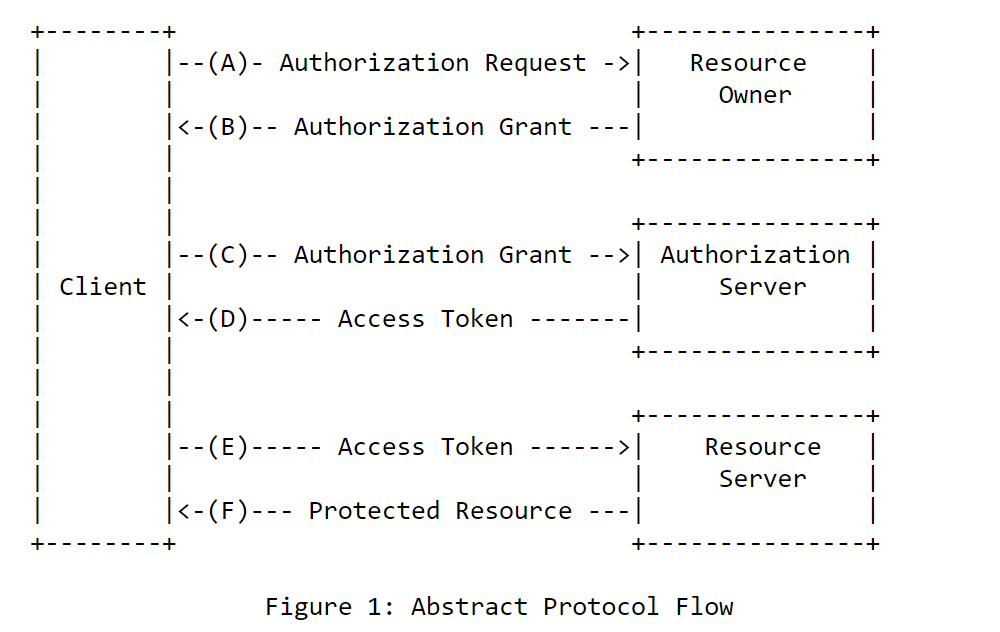
is beyond the scope of this specification. **The authorization server**

**may be the same server as the resource server or a separate entity**.

A single authorization server may issue access tokens accepted by

multiple resource servers.

### Protocol Flow



The abstract **OAuth 2.0** flow illustrated in Figure 1 describes the

interaction between the four roles and includes the following steps:

(A) **The client requests authorization from the resource owner**. The

authorization request can be made directly to the resource owner

(as shown), or preferably indirectly via the authorization

server as an intermediary.

**(B)** The client receives an **authorization grant**, which is a

**credential** **representing the resource owner's authorization**,

expressed using one of four grant types defined in this

specification or using an extension grant type. The

authorization grant type depends on the method used by the

client to request authorization and the types supported by the

authorization server.

(C) **The client requests an access token** by authenticating with the

authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates

the authorization grant, and if valid, **issues an access token**.

(E) The client requests the protected resource from the resource

server and authenticates by **presenting the access token**.

(F) **The resource server validates the access token**, and if valid,

serves the request.

The preferred method for the client to obtain an authorization grant

from the resource owner (depicted in steps (A) and (B)) is to **use the**

**authorization server as an intermediary**, which is illustrated in

Figure 3

### Authorization Grant

An authorization grant is a credential representing the resource

owner's authorization (to access its protected resources) used by the

client to obtain an access token. This specification defines **four**

**grant types** -- authorization code, implicit, resource owner password

credentials, and client credentials -- as well as an extensibility

mechanism for defining additional types.

#### [1.3.1](https://tools.ietf.org/html/rfc6749" \l "section-1.3.1). Authorization Code

The authorization code is obtained by using an authorization server

as an intermediary between the client and resource owner. **Instead of**

**requesting authorization directly from the resource owner**, **the client**

**directs the resource owner to an authorization server** (via its

**user-agent** as defined in [[RFC2616](https://tools.ietf.org/html/rfc2616)]), which in turn directs the

resource owner back to the client with the authorization code.

**Before directing the resource owner back to the client with the**

**authorization code, the authorization server authenticates the**

**resource owner and obtains authorization. Because the resource owner**

**only authenticates with the authorization server, the resource**

**owner's credentials are never shared with the client.**

The authorization code provides a few important security benefits,

such as the ability to authenticate the client, as well as the

transmission of the access token directly to the client without

passing it through the resource owner's user-agent and potentially

exposing it to others, including the resource owner.

#### [1.3.2](https://tools.ietf.org/html/rfc6749" \l "section-1.3.2). Implicit

The implicit grant is a **simplified authorization code flow** optimized

for clients implemented in a browser using a scripting language such

as JavaScript. In the implicit flow, **instead of issuing the client**

**an authorization code**, **the client is issued an access token directly**

Hardt Standards Track [Page 8]

[RFC 6749](https://tools.ietf.org/html/rfc6749) OAuth 2.0 October 2012

(as the result of the resource owner authorization). The grant type

is implicit, as no intermediate credentials (such as an authorization

code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the

authorization server does not authenticate the client. In some

cases, the client identity can be verified via the redirection URI

used to deliver the access token to the client. The access token may

be exposed to the resource owner or other applications with access to

the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some

clients (such as a client implemented as an in-browser application),

since it reduces the number of round trips required to obtain an

access token. However, this convenience should be weighed against

the security implications of using implicit grants, such as those

described in Sections [10.3](https://tools.ietf.org/html/rfc6749#section-10.3) and [10.16](https://tools.ietf.org/html/rfc6749#section-10.16), especially when the

authorization code grant type is available.

#### [1.3.3](https://tools.ietf.org/html/rfc6749" \l "section-1.3.3). Resource Owner Password Credentials

The resource owner **password credentials (i.e., username and password)**

can be used directly as an authorization grant to obtain an access

token. The credentials should only be used when there is a high

degree of trust between the resource owner and the client (e.g., the

client is part of the device operating system or a highly privileged

application), and when other authorization grant types are not

available (such as an authorization code).

Even though this grant type requires direct client access to the

resource owner credentials, the resource owner credentials are used

for a single request and are exchanged for an access token. This

grant type can eliminate the need for the client to store the

resource owner credentials for future use, by exchanging the

credentials with a long-lived access token or refresh token.

#### [1.3.4](https://tools.ietf.org/html/rfc6749" \l "section-1.3.4). Client Credentials

The client credentials (or other forms of client authentication) can

be used as an authorization grant when the authorization scope is

limited to the protected resources under the control of the client,

or to protected resources previously arranged with the authorization

server. Client credentials are used as an authorization grant

typically when the client is acting on its own behalf (the client is

also the resource owner) or is requesting access to protected

resources based on an authorization previously arranged with the

authorization server.

### [1.4](https://tools.ietf.org/html/rfc6749" \l "section-1.4). Access Token

Access tokens are credentials used to access protected resources. An

access token is a **string** representing an authorization issued to the

client. The string is usually opaque to the client. Tokens

represent **specific scopes** and **durations of access**, granted by the

resource owner, and enforced by the resource server and authorization

server.

The token may denote an identifier used to retrieve the authorization

information or may self-contain the authorization information in a

verifiable manner (i.e., a token string consisting of some data and a

signature). Additional authentication credentials, which are beyond

the scope of this specification, may be required in order for the

client to use a token.

The access token provides an abstraction layer, replacing different

authorization constructs (e.g., username and password) with a single

token understood by the resource server. This abstraction enables

issuing access tokens more restrictive than the authorization grant

used to obtain them, as well as removing the resource server's need

to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of

utilization (e.g., cryptographic properties) based on the resource

server security requirements. Access token attributes and the

methods used to access protected resources are beyond the scope of

this specification and are defined by companion specifications such

as [[RFC6750](https://tools.ietf.org/html/rfc6750)].

### [1.5](https://tools.ietf.org/html/rfc6749" \l "section-1.5). Refresh Token

Refresh tokens are **issued to the client by the authorization server and are**

**used to obtain a new access token when the current access token**

**becomes invalid or expires**, or to obtain additional access tokens

**with identical or narrower scope** (access tokens may have a shorter

lifetime and fewer permissions than authorized by the resource

owner). Issuing a refresh token is optional at the discretion of the

authorization server. If the authorization server issues a refresh

token, it is included when issuing an access token (i.e., step (D) in

Figure 1).

A refresh token is a string representing the authorization granted to

the client by the resource owner. The string is usually opaque to

the client. The token denotes an identifier used to retrieve the

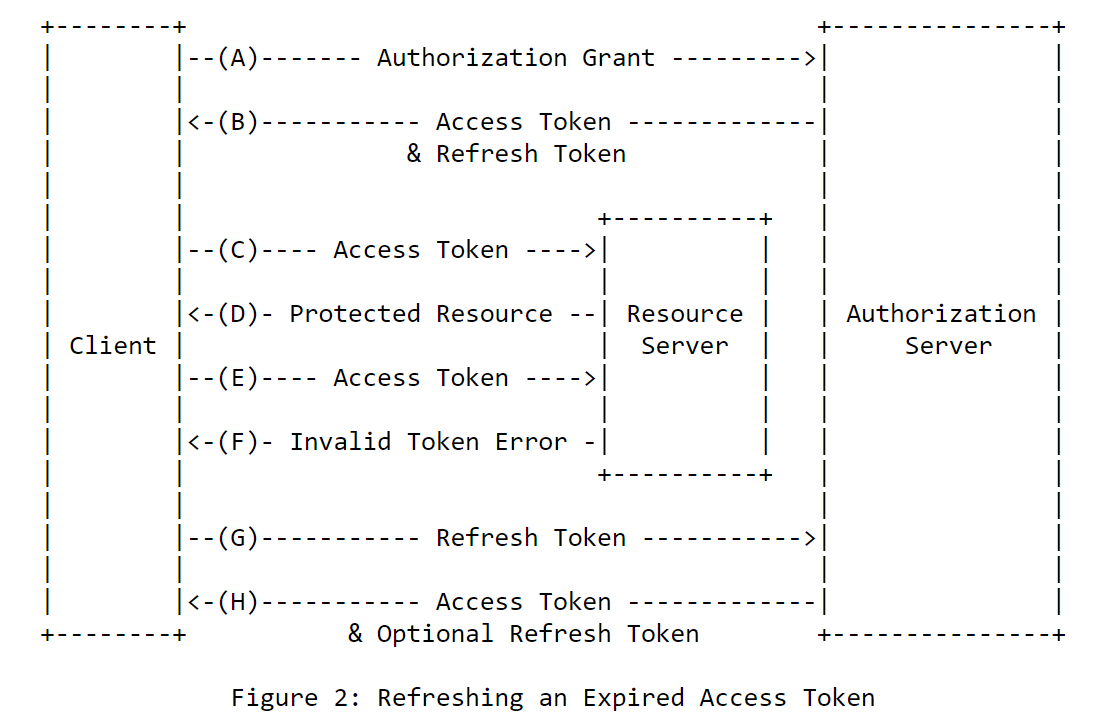
Hardt Standards Track [Page 10]

[RFC 6749](https://tools.ietf.org/html/rfc6749) OAuth 2.0 October 2012

authorization information. **Unlike access tokens, refresh tokens are**

**intended for use only with authorization servers and are never sent**

**to resource servers.**



The flow illustrated in Figure 2 includes the following steps:

(A) The client requests an access token by authenticating with the

authorization server and presenting an authorization grant.

(B) The **authorization** server authenticates the client and validates

the authorization grant, and if valid, **issues an access token**

**and a refresh token**.

(C) **The client makes a protected resource request to the resource**

**server by presenting the access token**.

(D) The **resource server validates the access token**, and if valid,

serves the request.

(E) Steps (C) and (D) repeat until **the access token expires**. If the

client knows the access token expired, it skips to step (G);

otherwise, it makes another protected resource request.

(F) Since the access token is invalid, the resource server returns

an invalid token error.

(G) The client **requests a new access token** by **authenticating** with

**the authorization server** and **presenting** the **refresh token**. The

client authentication requirements are based on the client type

and on the authorization server policies.

(H) The authorization server authenticates the client and validates

the refresh token, and if valid, **issues a new access token (and,**

**optionally, a new refresh token).**

**[2.1](https://tools.ietf.org/html/rfc6749" \l "section-2.1). Client Types**

OAuth defines **two client types**, based on their ability to

authenticate securely with the authorization server (i.e., ability to

maintain the confidentiality of their client credentials):

**confidential**

Clients capable of maintaining the confidentiality of their

credentials (e.g., client implemented on a secure server with

restricted access to the client credentials), or capable of secure

client authentication using other means.

**public**

Clients incapable of maintaining the confidentiality of their

credentials (e.g., clients executing on the device used by the

resource owner, such as an installed native application or a web

browser-based application), and incapable of secure client

authentication via any other means.

This specification has been designed around the following client

profiles:

**web application**

A web application is a confidential client **running on a web**

**server**. Resource owners access the client via an HTML user

interface rendered in a user-agent on the device used by the

resource owner. The client credentials as well as any access

token issued to the client are stored on the web server and are

not exposed to or accessible by the resource owner.

**user-agent-based application**

A user-agent-based application is a public client in which the

client code is downloaded from a web server and executes within a

**user-agent (e.g., web browser)** on the device used by the resource

owner. Protocol data and credentials are easily accessible (and

often visible) to the resource owner. Since such applications

reside within the user-agent, they can make seamless use of the

user-agent capabilities when requesting authorization.

**native application**

A native application is a public client installed and executed on

the device used by the resource owner. Protocol data and

credentials are accessible to the resource owner. It is assumed

that any client authentication credentials included in the

application can be extracted. On the other hand, dynamically

issued credentials such as access tokens or refresh tokens can

receive an acceptable level of protection. At a minimum, these

credentials are protected from hostile servers with which the

application may interact. On some platforms, these credentials

might be protected from other applications residing on the same

device.

* **Two authorization server endpoints**

(HTTP resources):

o **Authorization endpoint** - used by the client to obtain

authorization from the **resource owner** via user-agent redirection.

The authorization endpoint is used to interact with the **resource owner (Resource Server)** and **obtain an authorization grant**. The authorization server

MUST first verify the identity of the resource owner. The way in

which the authorization server authenticates the resource owner

(e.g., username and password login, session cookies) is beyond the

scope of this specification.

o **Token endpoint** - used by the client to exchange an authorization

grant for an access token, typically with client authentication.

**The token endpoint is used by the client to obtain an access token** by presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the

implicit grant type (since an access token is issued directly).

As well as one **client endpoint**:

o Redirection endpoint - used by the authorization server to return

responses containing authorization credentials to the client via

the resource owner user-agent.

Not every authorization grant type utilizes both endpoints.

Extension grant types MAY define additional endpoints as needed.

#### [4.1.3](https://tools.ietf.org/html/rfc6749" \l "section-4.1.3). Access Token Request

For example, the client makes the following HTTP request using TLS

(with extra line breaks for display purposes only):

**POST** /token HTTP/1.1

Host: server.example.com

**Authorization**: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

**grant\_type**=**authorization\_code**&**code**=SplxlOBeZQQYbYS6WxSbIA

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb

**[4.1.4](https://tools.ietf.org/html/rfc6749" \l "section-4.1.4). Access Token Response**

An example successful response:

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"**access\_token**":"2YotnFZFEjr1zCsicMWpAA",

"**token\_type**":"example",

"**expires\_in**":3600,

"**refresh\_token**":"tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter":"example\_value"

}

### [4.2](https://tools.ietf.org/html/rfc6749" \l "section-4.2). Implicit Grant

**GET** /authorize?response\_type=token&client\_id=s6BhdRkqt3&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1

Host: server.example.com

#### [4.2.2](https://tools.ietf.org/html/rfc6749" \l "section-4.2.2). Access Token Response

HTTP/1.1 302 Found

Location: http://example.com/cb#access\_token=2YotnFZFEjr1zCsicMWpAA

&state=xyz&token\_type=example&expires\_in=3600

### 

### [4.3](https://tools.ietf.org/html/rfc6749#section-4.3). Resource Owner Password Credentials Grant

POST /token HTTP/1.1

Host: server.example.com

**Authorization: Basic** czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

**grant\_type**=**password**&username=johndoe&password=A3ddj3w

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"2YotnFZFEjr1zCsicMWpAA",

"token\_type":"example",

"expires\_in":3600,

"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter":"example\_value"

}

## **[6](https://tools.ietf.org/html/rfc6749" \l "section-6). Refreshing an Access Token**

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

**grant\_type**=**refresh\_token**&refresh\_token=tGzv3JOkF0XG5Qx2TlKWIA

### [7.1](https://tools.ietf.org/html/rfc6749" \l "section-7.1). Access Token Types

The access token type provides the client with the information

required to successfully utilize the access token to make a protected

resource request (along with type-specific attributes). The client

MUST NOT use an access token if it does not understand the token

type.

For example, the "bearer" token type defined in [[RFC6750](https://tools.ietf.org/html/rfc6750)] is utilized

by simply including the access token string in the request:

**GET /resource/1 HTTP/1.1**

**Host: example.com**

**Authorization: Bearer mF\_9.B5f-4.1JqM**

while the "mac" token type defined in [[OAuth-HTTP-MAC](https://tools.ietf.org/html/rfc6749#ref-OAuth-HTTP-MAC)] is utilized by

issuing a Message Authentication Code (MAC) key together with the

access token that is used to sign certain components of the HTTP

requests:

**GET /resource/1 HTTP/1.1**

**Host: example.com**

**Authorization: MAC id="h480djs93hd8",**

**nonce="274312:dj83hs9s",**

**mac="kDZvddkndxvhGRXZhvuDjEWhGeE="**

## **[9](https://tools.ietf.org/html/rfc6749" \l "section-9). Native Applications**

**Native applications** are clients installed and executed on the device

used by the resource owner (i.e., **desktop application**, **native mobile**

**application**). Native applications require special consideration

related to security, platform capabilities, and overall end-user

experience.

The authorization endpoint requires interaction between the client

and the **resource owner's user-agent**. Native applications can invoke

an external user-agent or **embed a user-agent within the application**.

For example:

o **External user-agent** - the native application can capture the

response from the authorization server using a redirection URI

with a scheme registered with the operating system to invoke the

client as the handler, manual copy-and-paste of the credentials,

running a local web server, installing a user-agent extension, or

by providing a redirection URI identifying a server-hosted

resource under the client's control, which in turn makes the

response available to the native application.

o **Embedded user-agent** - the native application obtains the response

by directly communicating with the embedded user-agent by

monitoring state changes emitted during the resource load, or

accessing the user-agent's cookies storage.

When choosing between an external or embedded user-agent, developers

should consider the following:

o An external user-agent may improve completion rate, as the

resource owner may already have an active session with the

authorization server, removing the need to re-authenticate. It

provides a familiar end-user experience and functionality. The

Hardt Standards Track [Page 52]

[RFC 6749](https://tools.ietf.org/html/rfc6749) OAuth 2.0 October 2012

resource owner may also rely on user-agent features or extensions

to assist with authentication (e.g., password manager, 2-factor

device reader).

o An embedded user-agent may offer improved usability, as it removes

the need to switch context and open new windows.

o An embedded user-agent poses a security challenge because resource

owners are authenticating in an unidentified window without access

to the visual protections found in most external user-agents. An

embedded user-agent educates end-users to trust unidentified

requests for authentication (making phishing attacks easier to

execute).

When choosing between the implicit grant type and the authorization

code grant type, the following should be considered:

o Native applications that use the authorization code grant type

SHOULD do so without using client credentials, due to the native

application's inability to keep client credentials confidential.

o When using the implicit grant type flow, a refresh token is not

returned, which requires repeating the authorization process once

the access token expires.

# **The OAuth 2.0 Authorization Framework**: Bearer Token

This specification describes how to use bearer tokens in HTTP

requests to access OAuth 2.0 protected resources.

OAuth enables clients to access protected resources by obtaining an

access token, which is defined in "The OAuth 2.0 Authorization

Framework" [[RFC6749](https://tools.ietf.org/html/rfc6749)] as "a string representing an access

authorization issued to the client", rather than using the resource

owner's credentials directly.

**Tokens are issued to clients by an authorization server with the**

**approval of the resource owner.** The client uses the access token to

access the protected resources hosted by the resource server. This

specification describes how to make protected resource requests when

the OAuth access token is a bearer token.

TLS is mandatory to implement and use with this

specification

**Bearer Token**

A security token with the property that any party in possession of

the token (a "bearer") can use the token in any way that any other

party in possession of it can. Using a bearer token does not

require a bearer to prove possession of cryptographic key material

(proof-of-possession).

**OAuth** provides a method for clients to access a protected resource on

behalf of a resource owner. In the general case, before a client can

access a protected resource, it must first obtain an **authorization**

**grant from the resource owner** and then **exchange the authorization**

**grant for an access token.** The access token represents the grant's

scope, duration, and other attributes granted by the authorization

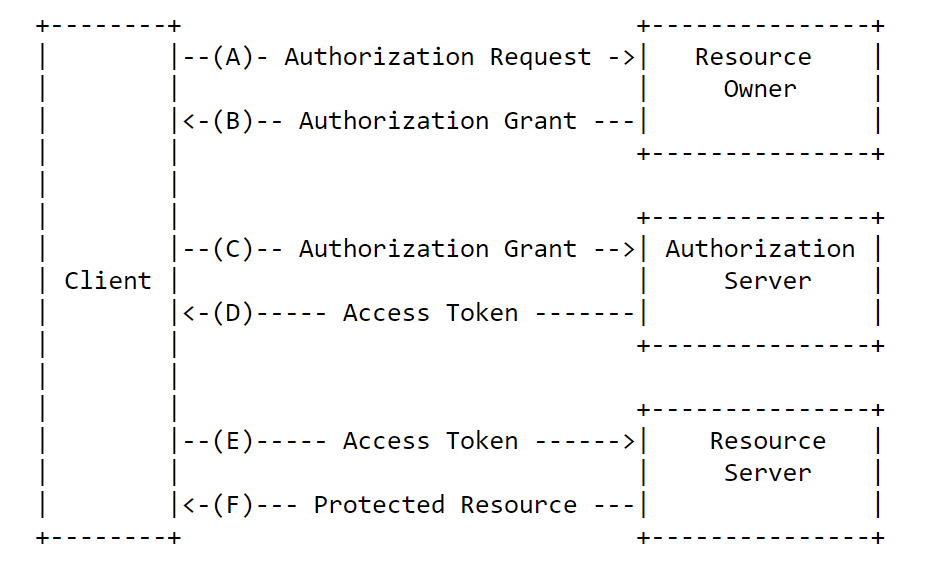
grant. The client accesses the protected resource by **presenting the**

**access token to the resource server**. **In some cases, a client can**

**directly present its own credentials to an authorization server to**

**obtain an access token without having to first obtain an**

**authorization grant from a resource owner**.



This section defines **three methods of sending bearer access tokens** in

resource requests to resource servers.

### Authorization Request Header

When sending the access token in the "Authorization" request header

field defined by HTTP/1.1 [[RFC2617](https://tools.ietf.org/html/rfc2617)], the client uses the **"Bearer"**

**authentication scheme** to transmit the access token.

For example:

**GET** /resource HTTP/1.1

Host: server.example.com

**Authorization: Bearer** mF\_9.B5f-4.1JqM

1. **Form-Encoded Body Parameter**

When sending the access token in the HTTP request entity-body, the

client adds the access token to the request-body using the

"access\_token" parameter.

The client MUST NOT use this method unless

all of the following conditions are met:

o The HTTP request entity-header includes the "**Content-Type**" header

field set to "**application/x-www-form-urlencoded**".

o The entity-body follows the encoding requirements of the

"application/x-www-form-urlencoded" content-type as defined by

HTML 4.01 [[W3C.REC-html401-19991224](https://tools.ietf.org/html/rfc6750#ref-W3C.REC-html401-19991224)].

o The HTTP request entity-body is single-part.

o The content to be encoded in the entity-body MUST consist entirely

of ASCII [[USASCII](https://tools.ietf.org/html/rfc6750#ref-USASCII)] characters.

o The HTTP request method is one for which the request-body has

defined semantics. In particular, this means that the "GET"

method MUST NOT be used.

For example, the client makes the following HTTP request using

transport-layer security:

**POST** /resource HTTP/1.1

Host: server.example.com

Content-Type: application/x-www-form-urlencoded

**access\_token=**mF\_9.B5f-4.1JqM

The "application/x-www-form-urlencoded" method **SHOULD NOT** be used

except in application contexts where **participating browsers do not**

**have access to the "Authorization" request header field**. Resource

servers MAY support this method.

### URI Query Parameter

When sending the access token in the HTTP request URI, the client

**adds the access token to the request URI query component** as defined

by "Uniform Resource Identifier (URI): Generic Syntax" [[RFC3986](https://tools.ietf.org/html/rfc3986)],

using the "access\_token" parameter.

For example, the client makes the following HTTP request using

transport-layer security:

**GET** /resource**?access\_token=**mF\_9.B5f-4.1JqM HTTP/1.1

Host: server.example.com

The HTTP request URI query can include other request-specific

parameters, in which case the "access\_token" parameter MUST be

properly separated from the request-specific parameters using "&"

character(s) (ASCII code 38).

For example:

https://server.example.com/resource**?**access\_token=mF\_9.B5f-4.1JqM**&**p=q

Clients using the URI Query Parameter method SHOULD also send a

**Cache-Control** header containing the "**no-store**" option. Server

success (2XX status) responses to these requests SHOULD contain a

**Cache-Control** header with the "**private**" option.

Because of the security weaknesses associated with the URI method

(see [Section 5](https://tools.ietf.org/html/rfc6750#section-5)), including the high likelihood that the URL

containing the access token will be logged, it SHOULD NOT be used

**unless** it is impossible to transport the access token in the

"Authorization" request header field or the HTTP request entity-body.

Resource servers MAY support this method.

This method is included to document current use; its use is not

recommended, due to its security deficiencies (see [Section 5](https://tools.ietf.org/html/rfc6750#section-5)) and

also because it uses a reserved query parameter name, which is

counter to URI namespace best practices, per "Architecture of the

World Wide Web, Volume One" [[W3C.REC-webarch-20041215](https://tools.ietf.org/html/rfc6750#ref-W3C.REC-webarch-20041215)].

For example, in response to a protected resource request **without**

**authentication**:

HTTP/1.1 **401 Unauthorized**

**WWW-Authenticate:** Bearer realm="example"

And in response to a protected resource request with an

authentication attempt using an **expired access token**:

HTTP/1.1 401 Unauthorized

**WWW-Authenticate: Bearer** realm="example",

**error="invalid\_token"**,

error\_description="The access token expired"

## **Example Access Token Response**

Typically, a bearer token is returned to the client as part of an

OAuth 2.0 [[RFC6749](https://tools.ietf.org/html/rfc6749)] access token response. An example of such a

response is:

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

**"access\_token":"mF\_9.B5f-4.1JqM",**

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

**"expires\_in":3600,**

**"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA"**

**}**

The point of having access tokens is that they can be used without checking for invalidation. You can have 10000 frontend servers users can access with the token without the need to ever ask some database if it is invalid. But after some time, the token expires. The user needs a new access token, sends her refresh token and **this refresh token is checked in some database**. You never need to check for expired access tokens or have any state, but limit abuse to the lifetime of the token.

If you don't have the requirement to accept the tokens without checking expiration in a database, you don't need the two different tokens. You can just use the refresh token for each access.

**Example login workflow** would be:

1. **User logs in**, gets access and refresh token. Access token lifetime 15min, refresh token 5 days.
2. User accesses the service using the access token. Service only checks signature and lifetime. No database connection.
3. **User logs out, the refresh token** is marked expired in the **database**
4. User accesses the service using the access token, this still works
5. 15min pass. Access token expires, user requests a new access token using the refresh token still within its lifetime. The service checks the **database** and finds the access token is expired. User can't get a new access token.

The whole system is a trade-off between the time it takes to invalidate sessions and the amount of connections to a shared/synchronized data source required.

You can replace the refresh token on each refresh, but remember that you need to **store all expired refresh tokens until their lifetime is over**.

From a security perspective it makes sense to create a new token, but it is a trade off between security and amount of data in your database.

In the worst case (no lifetime for refresh tokens, **never** do that) you now need to store one token every few minutes in the database for each user instead of one token for each user and you can never remove them again.

expires\_in: RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

## **Token Refresh Handling: Method 1**

Upon receiving a valid access\_token, expires\_in value, refresh\_token, etc., clients can process this by storing an expiration time and checking it on each request. This can be done using the following steps:

1. **convert expires\_in to an expire time (epoch, RFC-3339/ISO-8601 datetime, etc.)**
2. **store the expire time**
3. **on each resource request, check the current time against the expire time and make a token refresh request before the resource request if the access\_token has expired**

An example implementation is the Go oauth2 library which converts the expires\_in value to a RFC 3339 date-time in the Token [expiry property](https://godoc.org/golang.org/x/oauth2#Token). expiry isn't defined by the OAuth 2.0 standard but is useful here.

When checking the time, be sure you are the same time, for example, using the same timezone by converting all times to epoch or UTC timezone.

In addition to receiving a new access\_token, you may receive a new refresh\_token with an expiration time further in the future. If you receive this, you should store the new refresh\_token to extend the life of your session.

## **Token Refresh Handling: Method 2**

Another method of handling token refresh is to manually refresh after receiving an invalid token authorization error. This can be done with the previous approach or by itself.

If you attempt to use an expired access\_token and you get an invalid token error, you should perform a token refresh (if your refresh token is still valid). Since different services can use different error codes for expired tokens, you can either keep track of the code for each service or an easy way to refresh tokens across services is to simply try a single refresh upon encountering a 4xx error.

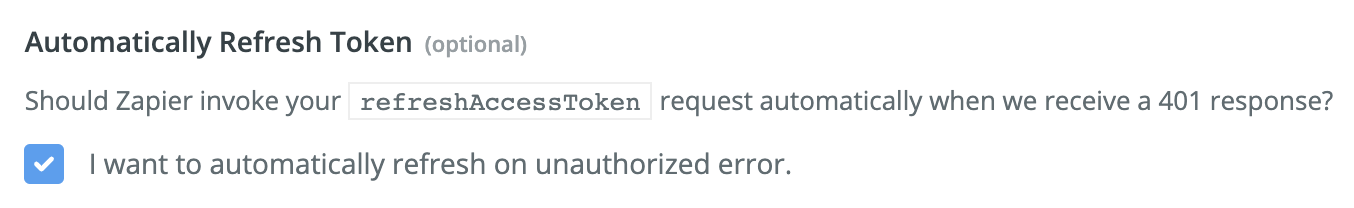
### Invalid Access Token Errors

Below are some error codes from popular services:

1. [Facebook: Error 467 Invalid access token](https://developers.facebook.com/docs/graph-api/using-graph-api/v2.3#errors) - Access token has expired, been revoked, or is otherwise invalid - Handle expired access tokens.
2. [LinkedIn: Error 401 Unauthorized](https://developer.linkedin.com/docs/oauth2).
3. [PayPal: Error 401 Unauthorized](https://developer.paypal.com/docs/integration/direct/paypal-oauth2/).

### Implementations

The Zapier service is one service that implements the refresh after authorization error retry.

[](https://i.stack.imgur.com/P8K0z.png)

## **Refresh Token Expiration**

**If your refresh\_token has also expired, you will need to go through the authorization process again**.

The [OAuth 2.0 spec](https://tools.ietf.org/html/rfc6749) doesn't define refresh token expiration or how to handle it, however, a number of APIs will return a refresh\_token\_expires\_in property when the refresh token does expire. Different APIs will handle refresh token expiration differently so it's important to review the docs per API, but generally you may receive a new refresh token when you refresh your access token. Expiration should be handled in a similar way such as converting refresh\_token\_expires\_in to a RFC 3339 date-time refresh\_token\_expiry value.

Some examples include [LinkedIn](https://developer.linkedin.com/docs/Refresh-Tokens-with-OAuth-2), [eBay](https://developer.ebay.com/api-docs/static/oauth-qref-auth-code-grant.html), and [RingCentral](https://developers.ringcentral.com/api-reference). In the LinkedIn API, when you refresh access tokens, you will receive a refresh token with a decreasing refresh\_token\_expires\_in property targeting the original refresh token expiry time until you are required to auth again. The RingCentral API will return refresh tokens with a static time so the user does not have to auth again if token refreshes and refresh token updates are done consistently.

<https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/>

With JWTs, a "logout" is simply deleting the token on the client side so that it can't be used for subsequent API calls.

A **logout endpoint is not really required**, because **any microservice that accepts your JWTs will keep accepting it**. If your auth server deletes the JWT, it won't matter because the other services will keep accepting it anyway (since the whole point of JWTs was to **not require centralised coordination**).

**The token is still valid and can be used. What if I need to ensure that the token cannot be used ever again?**

This is why keeping JWT expiry values to a small value is important. And this is why ensuring that your JWTs don't get stolen is even more important. The token is valid (even after you delete it on the client), but only for short period to reduce the probability of it being used maliciously.

In addition, you can add a **deny-listing** workflow to your JWTs. In this case, you can have a /logout API call and your auth server puts the tokens in a "invalid list". However, all the API services that consume the JWT now need to add an additional step to their JWT verification to check with the centralized "deny-list". **This introduces central state** again, and brings us back to what we had before using JWTs at all.

**Doesn’t deny-listing negate the benefit of JWT not needing any central storage?**

In a way it does. It’s an optional precaution that you can take if you are worried that your token can get stolen and misused, but it also increases the amount of verification that has to be done. As you can imagine, this had led to much [gnashing of teeth on the internet](https://stackoverflow.com/questions/21978658/invalidating-json-web-tokens/52407314#52407314).

**What will happen if I am logged in on different tabs?**

One way of solving this is by introducing a global event listener on localstorage. Whenever we update this logout key in localstorage on one tab, the listener will fire on the other tabs and trigger a "logout" too and redirect users to the login screen.

In that case whenever you log out from one tab, event listener will fire in all other tabs and redirect them to login screen.

**This works across tabs. But how do I "force logout" of all sessions on different devices?!**

**Silent refresh**

There are 2 major problems that users of our JWT based app will still face:

1. Given our short expiry times on the JWTs, the user will be logged out every 15 minutes. This would be a fairly terrible experience. Ideally, we'd probably want our user to be logged in for a long time.
2. If a user closes their app and opens it again, they'll need to login again. Their session is not *persisted*because we're not saving the JWT token on the client anywhere.

To solve this problem, most JWT providers, provide a refresh token. A refresh token has 2 properties:

1. It can be used to make an API call (say, /refresh\_token) to fetch a new JWT token before the previous JWT expires.
2. It can be safely persisted across sessions on the client!
3. **How does a refresh token work?**

This token is issued as part of authentication process along with the JWT. The **auth server** should **saves this refresh token and associates it to a particular user in its own database**, so that it can handle the renewing JWT logic.

1. On the client, before the previous JWT token expires, we wire up our app to make a /refresh\_token endpoint and grab a new JWT.

**How is a refresh token safely persisted on the client?!**

The refresh token is sent by the auth server to the client as an HttpOnly cookie and is automatically sent by the browser in a /refresh\_token API call.

Because client side Javascript can't read or steal an HttpOnly cookie, this is a little better at mitigating XSS than persisting it as a normal cookie or in localstorage.

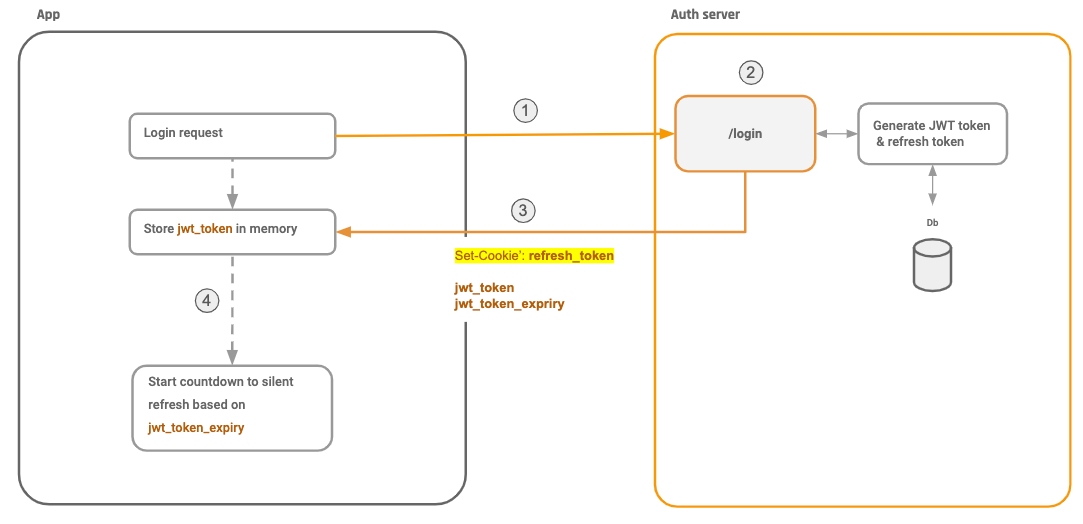
This is safe from [CSRF](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) attacks, because even though a form submit attack can make a /refresh\_token API call, the attacker cannot get the new JWT token value that is returned.  
  
To recap, this is how we're thinking about what would be the best way of persisting a JWT based session:?

Persisting JWT token in localstorage (prone to XSS) < Persisting JWT token in an HttpOnly cookie (prone to CSRF, a little bit better for XSS) < Persisting refresh token in an HttpOnly cookie (safe from CSRF, a little bit better for XSS).

Note that while this method is not resilient to serious XSS attacks, coupled with the [usual XSS mitigation techniques](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules), an HttpOnly cookie is a [recommended way](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#bonus-rule-1-use-httponly-cookie-flag) persisting session related information. But by persisting our session indirectly via a refresh token, we prevent a direct CSRF vulnerability we would have had with a JWT token.

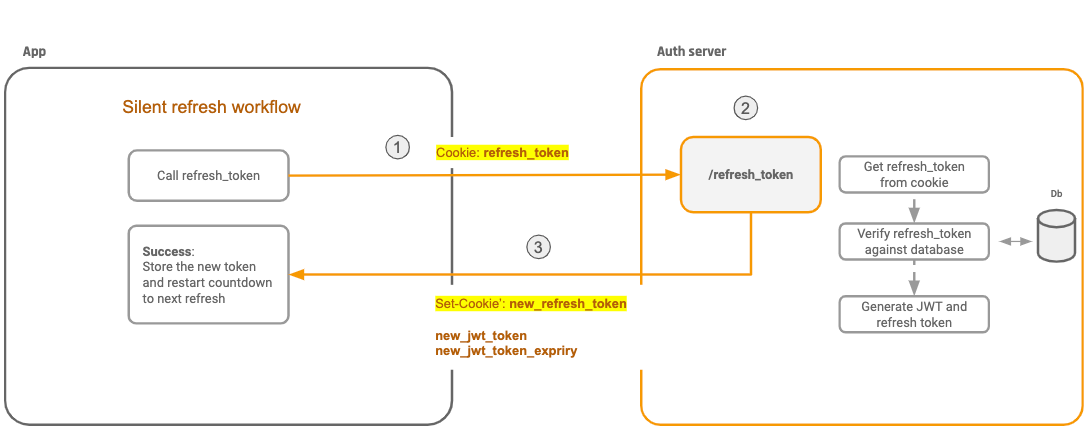
**So what does the new "login" process look like?**

Nothing much changes, except that a refresh token gets sent along with the JWT. Let's take a look a diagram of login process again, but now with refresh\_token functionality:

**Login with refresh token**

1. The user logs in with a login API call.
2. Server generates JWT Token and refresh\_token
3. Server sets a HttpOnly cookie with refresh\_token. jwt\_token and jwt\_token\_expiry are returned back to the client as a JSON payload.
4. The jwt\_token is stored in memory.
5. A countdown to a future silent refresh is started based on jwt\_token\_expiry

**And now, what does the silent refresh look like?**

Silent refresh workflow

Here's what happens:

1. Call /refresh\_token endpoint
2. Server will read httpOnly cookie and if it finds a valid refresh\_token, then...
3. ...the server returns a new jwt\_token and jwt\_token\_expiry to the client and also sets a new *refresh token* cookie via  Set-Cookie header.

where the connection between user and either of them is (usually) equally secure, there is not much sense to keep refresh token separate from the access token.

Although, as mentioned in the quote, another role of refresh tokens is to ensure the access token can be revoked at any time by the User (via the web-interface in their profiles, for example) while keeping the system scalable at the same time.

Generally, tokens can either be random identifiers pointing to the specific record in the Server's database, or they can contain all information in themselves (certainly, this information have to be signed, with [MAC](http://en.wikipedia.org/wiki/Message_authentication_code), for example).

**How the system with long-lived access tokens should work**

The server allows the Client to get access to User's data within a pre-defined set of scopes by issuing a token. **As we want to keep the token revocable, we must store in the database the token along with the flag "revoked" being set or unset** (otherwise, how would you do that with self-contained token?) Database can contain as much as len(users) x len(registered clients) x len(scopes combination) records. Every API request then must hit the database. Although it's quite trivial to make queries to such database performing O(1), the single point of failure itself can have negative impact on the scalability and performance of the system.

**How the system with long-lived refresh token and short-lived access token should work**

Here we issue two keys: random **refresh token** with the corresponding record **in the database**, and signed self-contained access token, containing among others the **expiration** timestamp field.

As the access token is self-contained, we don't have to hit the database at all to check its validity. All we have to do is to decode the token and to validate the signature and the timestamp.

Nonetheless, we still have to keep the database of refresh tokens, but the number of requests to this database is generally defined by the lifespan of the access token (the longer the lifespan, the lower the access rate).

In order to revoke the access of Client from a particular User, we should mark the corresponding refresh token as "revoked" (or remove it completely) and stop issuing new access tokens. It's obvious though that there is a window during which the refresh token has been revoked, but its access token may still be valid.

**Tradeoffs**

Refresh tokens partially eliminate the SPoF (Single Point of Failure) of Access Token database, yet they have some obvious drawbacks.

1. The "window". A timeframe between events "user revokes the access" and "access is guaranteed to be revoked".
2. The complication of the **Client** logic.

**without** refresh token

* + send API request with access token
  + if access token is invalid, fail and **ask user to re-authenticate**

**with** refresh token

* + send API request with access token
  + If access token is invalid, try to update it using refresh token
  + if refresh request passes, update the access token and re-send the initial API request
  + If refresh request fails, ask user to re-authenticate

I hope this answer does make sense and helps somebody to make more thoughtful decision. I'd like to note also that some well-known OAuth2 providers, including github and foursquare adopt protocol without refresh tokens, and seem happy with that.

##

I am building my own authentication microservice and although I have the main setup in place (generating access tokens etc.), I am a bit lost when it comes to refresh tokens.

I feel there are a lot of different way to handle this.

* You can either store them in Redis or in the database.
* You can use a whitelist or a blacklist them

Right now, my idea is to add another database table that links a valid refresh token to a user entity. When a user hits the logout endpoint, the refresh token gets destroyed.

I was wondering if this was a good solution and otherwise, if there are other possible solutions to consider. I have seen a number of articles when googling but they stem from anywhere between 2015 and 2019, and they all have different approaches.

The issue with refresh tokens is not so much where or how you store them on the server side, as well if and how you store them on the client side.

It all depends on whether you can trust your client (software using the token) to keep secrets. You only want to issue refresh tokens to a client you can trust to keep these tokens secure. Typically, this means only issue refresh tokens to [**confidential clients**](https://tools.ietf.org/html/rfc6749#section-3.2.1), i.e. web applications that run on a web server. These clients can also have their own (client) credentials to authenticate themselves with when using the refresh token.

For **public (non-confidential) clients**, like Single Page Applications, some OAuth2 libraries use an hidden IFRAME and a cookie session with the authorization server to issue new access tokens.

So, the answer to your question depends on what kind of clients you will be using.

I'm getting started with **token based authentication** using **the ASOS (AspNet.Security.OpenIdConnect.Server) framework.**

I've got the access token generation and retrieval done and am now moving on to the refresh token bit.

**My questions are**:

* How should I store the refresh token server side?
  + Should I just store the **clientID and the hashed and salted refresh token in a database** (Along with utility fields, such as an expiration date)?
* What is the expected behavior if a user of my API has a single clientID and secret, but performs many calls concurrently (Suppose they want to scale out the client on their end across multiple machines to get better throughput for example).
  + **Specifically, I mean what if 1 of the client's access tokens expires, but their refresh token has also expired? Of course they can go to the token endpoint to get a new access token and refresh token at the same time, but then what about the other instances for that clientID? Assuming that their code is identical (i.e. they don't share knowledge of the refresh token), each instance will also go on to request a new access and refresh token.**
  + **If you store a single refresh token for a clientID, you'll end up excessively requesting refresh tokens, potentially every time the access token expires, which would be undesirable.**

**Google seems to only allow a single refresh token, so I'll go the route of just allowing one and make it the consumer's task to manage the sharing of the refresh token within their app**.

* + If you store multiple refresh tokens for a client, how many is a sensible number?

Also, what is the common process of revoking the refresh tokens? Is it as simple as just deleting it from wherever you're storing it?

It really depends on your application requirements and how you implement that. You're free to consider refresh tokens as completely independent or, conversely, interdependent. This logic would usually take place into HandleTokenRequest.

If you use the default token format (more than recommended), refresh tokens will be considered valid until they expire. It's up to you to check whether the token has been revoked from HandleTokenRequest by making a **DB lookup** (you can get the refresh token identifier using context.Ticket.GetTokenId())

###

[Token authentication](https://stormpath.com/product/token-auth/) is stateless, secure, Scalable, mobile-ready, and designed to grow with your user base without adding additional strain on your servers.