**OAuth 2.0, OpenID Connect y JSON Web Tokens - PCKE**

Para llevar a cabo algunas pruebas utilizaremos: Auth0

<https://manage.auth0.com/dashboard/us/dev-utn-frc-iaew/>

<https://www.getpostman.com/collections/cfbb713a3f5c4029ad1e>

|  |
| --- |
| Ver: **Links útiles:** <https://www.paradigmadigital.com/dev/oauth-2-0-equilibrio-y-usabilidad-en-la-securizacion-de-apis/>  <https://www.websecurity.digicert.com/es/es/security-topics/what-is-ssl-tls-https> |

|  |
| --- |
| ¿Qué es Auth0? [https://manage.auth0.com](https://manage.auth0.com/dashboard/us/dev-utn-frc-iaew/)  Auth0 es una plataforma de nivel empresarial para la identidad moderna. Autentica y autoriza aplicaciones y API con cualquier proveedor de identidad popular (empresarial, social o personalizado) que se ejecute en cualquier conjunto de software popular en un dispositivo o nube popular. Totalmente ampliable, las características de Auth0 están expuestas como una API, y cada parte del canal de autenticación y autorización se puede personalizar.  Artículo Interesante: <https://www.forbesargentina.com/la-increible-historia-de-como-auth0-se-convirtio-en-el-quinto-unicornio-argentino/> |

Podes crear tu cuenta gratuita ingresando a <https://manage.auth0.com/>

**¿Por qué aparecen estas tecnologías?**

Aparecen por la necesidad de poder acceder a un **recurso (API)** en tu nombre. Puede que ese servicio pertenezca a la compañía en la que trabajas o quizás a otra donde eres cliente/usuario. Los escenarios u objetivos que se quieren alcanzar pueden ser de todo tipo. Estos son algunos ejemplos, para que te hagas una idea:

* Hacer una publicación en tu nombre en una red social (en Twitter, Facebook, Instagram, etcétera).
* Mandar un correo electrónico a alguien en tu nombre.
* Recuperar tus movimientos bancarios.
* Lo que se te ocurra, pero en tu nombre 🙂

Sin embargo, es importante que sepas diferenciar que **no es lo mismo saber quién eres (autenticación) que, qué es lo que puedes hacer (autorización)**. Aquí entra en juego la autenticación y la autorización.

**Autenticación y autorización**

Antes de hablar de OAuth 2.0 y OpenID Connect es importante que tengas claro qué es la autenticación y la autorización.

**La autenticación es el proceso de verificar una identidad**, es decir confirmar que **una persona es quien dice ser**. Normalmente, para constatar este hecho, el usuario usa algo que conoce para demostrar su identidad, como un usuario y una contraseña.

Por otro lado, **la autorización es el proceso de verificar lo que un usuario puede hacer**. Por ejemplo, un usuario puede añadir canciones en una lista compartida de Spotify, pero no puede eliminar dicha lista. La autorización ocurre después de que un usuario se haya autenticado.

**Métodos de autenticación y autorización antes de OAuth**

Para entender por qué apareció OAuth, creo que es importante que conozcas qué usábamos antes de este, y así comprender su necesidad. Los métodos más comunes para autenticarte y gestionar la autorización en las APIs era compartiendo directamente el usuario y la contraseña (Credential Sharing), usando cookies y a través de API Keys. Existen más opciones, claro está, pero estas son las más comunes o las más extendidas.

**Compartir el usuario y la contraseña**

Hasta hace un tiempo atrás, la forma de llevar a cabo la autenticación era compartiendo el usuario y la contraseña con el servicio que necesita actuar en tu nombre. El problema en este escenario es que estamos actuando exactamente como usuario final, lo cual se llama **impersonation**(interpretación en español queda un poco raro). En este caso, no tenemos forma de restringir el acceso a ciertas partes de nuestra API, es decir, no podemos delegar solo ciertos permisos a la aplicación que está actuando por nosotros. Imagina a día de hoy, donde infinidad de aplicaciones quieren llevar a cabo algunas acciones con tu perfil de Facebook, normalmente publicar en tu muro y obtener tu lista de amigos.

Si utilizáramos este sistema le daríamos acceso a absolutamente todo, y podría recuperar conversaciones privadas, nuestras fotos, seguir a gente que no queremos, eliminar amigos que si queremos… ¿te haces una idea no?

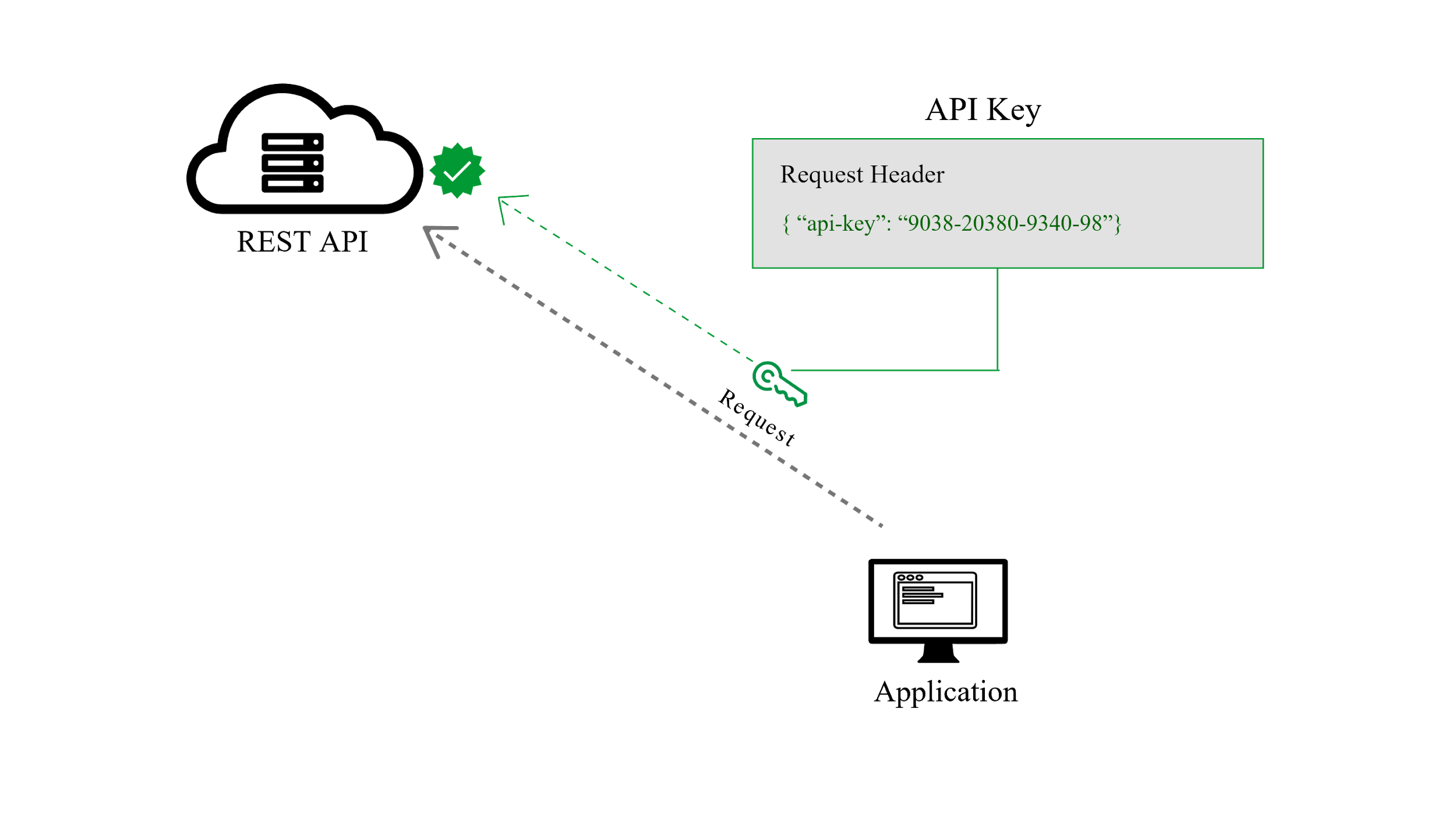
**Cookies**

En este caso, si la autenticación ha sido satisfactoria, el sistema nos dará una cookie. Esta permitirá el acceso al recurso o API. El problema principal en el uso de estas es el Cross-Site Request Forgery (CSRF o XSRF). Si el usuario abre otro tab en el navegador y el usuario accede al mismo sitio te darás cuenta de que puedes seguir accediendo sin problemas. Lo que significa es que le estamos dando acceso al navegador entero, no solo a nuestro tab donde está la aplicación web que consulta nuestra información. Por lo que un atacante podría inyectar código en nuestro HTML que llamara a nuestro servicio y, sin nuestro consentimiento, podría llevar a cabo tareas en nuestro nombre.

**API Keys**

Las API Keys son el primer método que apareció para realizar accesos a este tipo de servicios. Los desarrolladores no dudan en pensar en ella como la primera alternativa a la hora de autenticarse en su API. Esto se debe a que entre sus ventajas están su simpleza, la posibilidad de sacar estadísticas de accesos o peticiones y a su facilidad de uso para implementarlas en las aplicaciones que consumirá el servicio. Generalmente cuando accedemos a una web con este tipo de autenticación, vamos a nuestro perfil y en la sección de configuración nos encontramos con la API Key lista para ser cargada en alguna app móvil o servicio web para hacer tests de API.

Todo esto trae aparejados dos grandes problemas que son: su falta de seguridad y la pésima experiencia de usuario. En muchos casos, estas claves son enviadas dentro de la URL en donde se realiza la petición al servicio, lo que permite ser accedida por alguien que no debería. Para esto el estándar nos dice que las keys deberían viajar en el Header de la petición, para dificultar un poco su acceso por parte de terceros, aunque en la práctica se pueden ver en el Header Authorization, en un Header personalizado o en la URL.

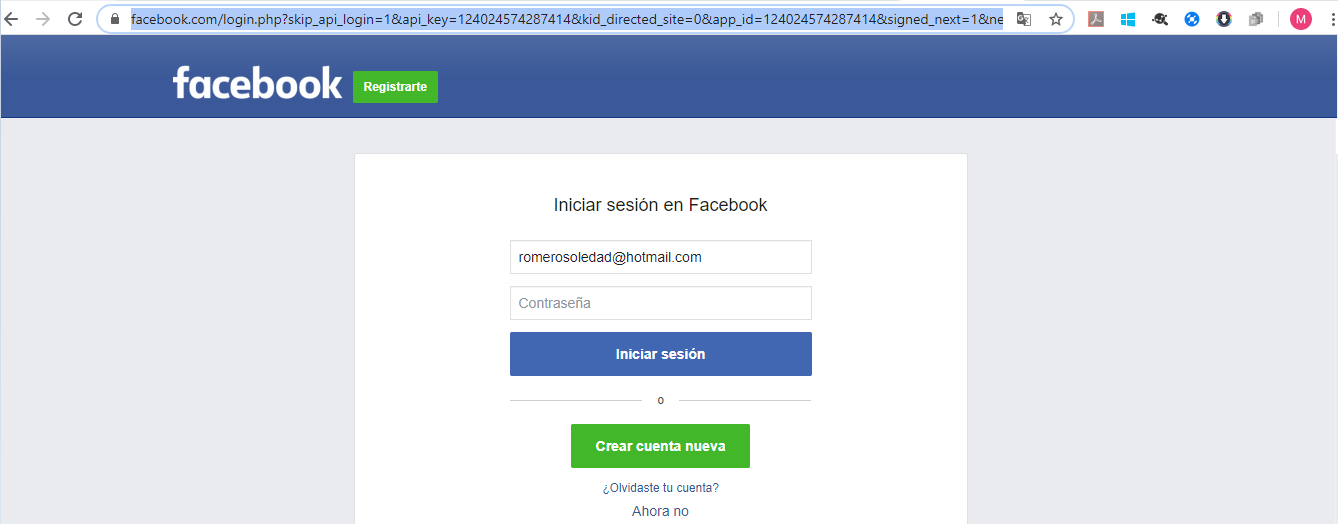


Ejemplo:

Instagram <https://www.instagram.com/>

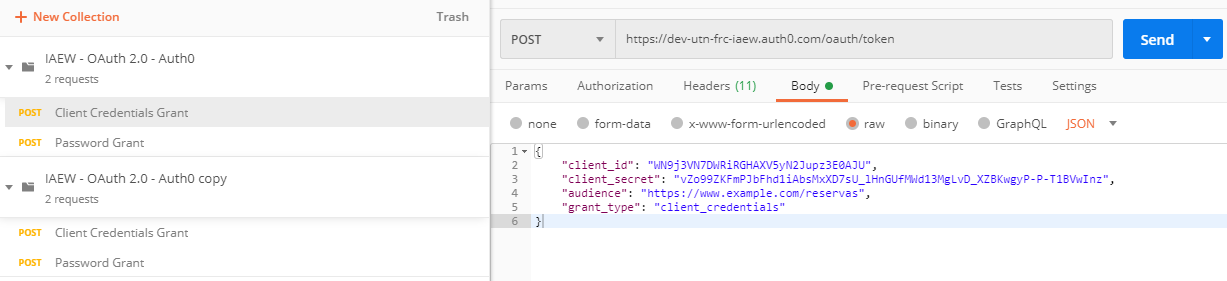
Al seleccionar Iniciar sesión con facebook…

<https://www.facebook.com/login.php?skip_api_login=1&api_key=124024574287414&kid_directed_site=0&app_id=124024574287414&signed_next=1&next=https%3A%2F%2Fwww.facebook.com%2Fdialog%2Foauth%3Fclient_id%3D124024574287414%26redirect_uri%3Dhttps%253A%252F%252Fwww.instagram.com%252Faccounts%252Fsignup%252F%26state%3D%257B%2522fbLoginKey%2522%253A%252210sg12jopltt21s7lr451eixkr0e8hz21ai1jhgbfyv4s113l2pc%2522%252C%2522fbLoginReturnURL%2522%253A%2522%252F%2522%257D%26scope%3Demail%26response_type%3Dcode%252Cgranted_scopes%26locale%3Des_LA%26ret%3Dlogin%26fbapp_pres%3D0%26logger_id%3D67335f4d-e66b-4dbf-a195-6ade6e01dd07&cancel_url=https%3A%2F%2Fwww.instagram.com%2Faccounts%2Fsignup%2F%3Ferror%3Daccess_denied%26error_code%3D200%26error_description%3DPermissions%2Berror%26error_reason%3Duser_denied%26state%3D%257B%2522fbLoginKey%2522%253A%252210sg12jopltt21s7lr451eixkr0e8hz21ai1jhgbfyv4s113l2pc%2522%252C%2522fbLoginReturnURL%2522%253A%2522%252F%2522%257D%23_%3D_&display=page&locale=es_LA&pl_dbl=0>



**OAuth 2**

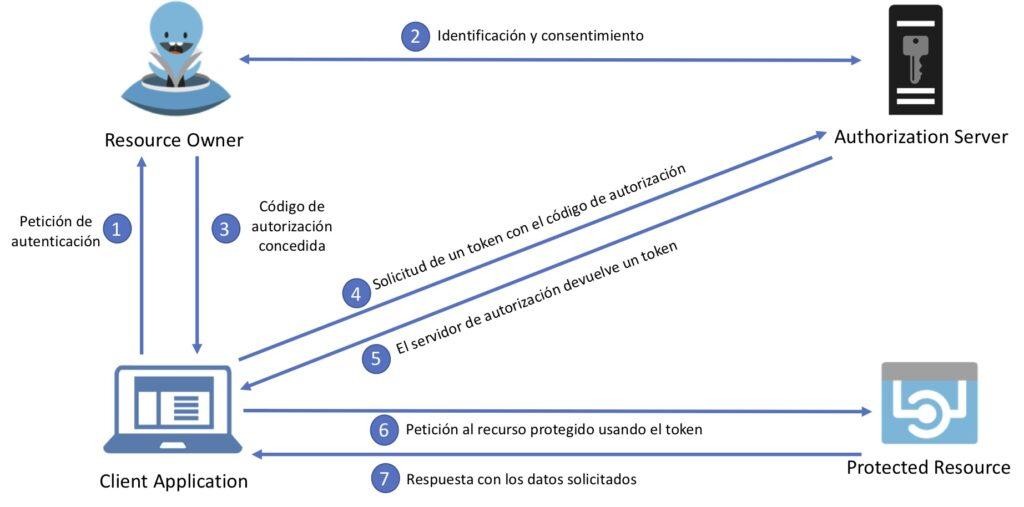
COLECCION POSTMAN: <https://www.getpostman.com/collections/80f72912aade0c4a4590>



OAuth se construyó específicamente para acceder a APIs a través de HTTP. **El usuario delega en la aplicación la capacidad de realizar ciertas acciones en su nombre**. Es importante recalcar que OAuth es un framework para la autorización, que no la autenticación.  
  
En OAuth 2.0 se trabaja con estas 4 partes:

* **Protected Resource:**El recurso al que queremos acceder, una API.
* **Client:**La aplicación que quiere acceder al recurso que está protegido, en nombre de alguien. Este cliente puede ser una aplicación web, móvil, de escritorio, para Smart TV, un dispositivo IoT, etcétera.
* **Resource Owner**: Se trata del usuario. Se le llama el propietario de los recursos porque, si bien la API no es tuya los datos que maneja si lo son.
* **Authorization server**: es el responsable de gestionar las peticiones de autorización.

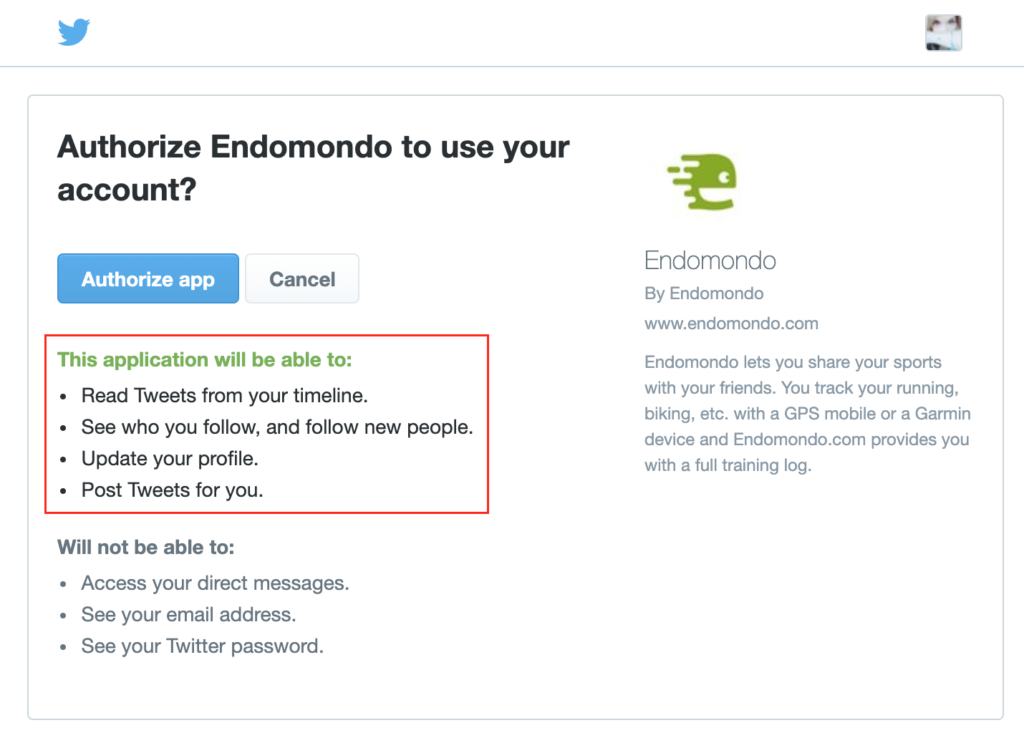
La forma en la que estas 4 partes se relacionan entre sí es la siguiente:

Cómo funciona OAuth 2.0

1. La aplicación solicita al usuario que se autentique.
2. Para ello, este es redirigido al servidor de autorización para su identificación. Este puede loguearse a través de usuario y contraseña, a través de reconocimiento facial, de voz, puede tener un segundo factor de autenticación o incluso se le puede permitir el acceso a través de otras cuentas que son del usuario, como Google, Facebook, Amazon, etcétera. Lo que sea necesario para verificar su identidad. Una vez que el usuario es validado, este debe estar de acuerdo con lo que la aplicación quiere hacer (esto se llama consentimiento). Normalmente se muestra una lista de los permisos que la aplicación está solicitando.
3. Cuando la identidad del usuario es validada de manera satisfactoria, y este ha consentido lo que se quiere hacer con esta autorización, el usuario es redirigido de nuevo a la aplicación cliente. Le otorga a esta un código confirmando que después de que el usuario se ha autenticado, este le autoriza hacer cosas por él en el recurso protegido.
4. La aplicación cliente hace una solicitud al servidor de autorización, con el permiso que el usuario le ha dado, además de algunos datos que identifican a la aplicación para que se verifique que es un cliente válido para acceder a los recursos que se propone.
5. Si todo va bien, el servidor de autorización responderá con un token de acceso (access token en inglés). Access\_token
6. Con este access token la aplicación cliente será capaz de realizar llamadas a la API que necesita acceder para llevar a cabo su cometido.
7. Cuando el recurso protegido recibe el access token necesita verificarlo de alguna forma. Una vez validado, comprobará si el usuario tiene los permisos por los cuales se hizo la petición y, si está todo ok, la API responderá con los datos que se le han pedido.

Ahora que ya has comprendido la foto general, vamos a ver algunos detalles de este flujo.

**Consentimiento (Consent)**

En el paso 2 el usuario debe “consentir” lo que la aplicación está pidiendo de él. Twitter pide el consentimiento del usuario para otorgarle a Endomodo permisos.

En OAuth 2.0 te aseguras de que el usuario es consciente de para qué tu aplicación quiere tu permiso y qué puede hacer con él.

**Endpoints**

Para el paso 2 y el paso 4, en OAuth tenemos dos Endpoints o URLs en nuestro servidor de autorización:

* **Authorization endpoint (/authorize)**: se utiliza para la interacción con usuarios, cuando este tiene que identificarse.
* **Token endpoint (/token)**: sólo para máquinas, sin interacción del usuario.

Ambos usan TLS (Transport Security Layer) y estas URLs deben ser conocidas por la aplicación cliente (<https://www.websecurity.digicert.com/es/es/security-topics/what-is-ssl-tls-https>)

**Scopes**

Los scopes son los permisos para hacer algo dentro de un recurso protegido en nombre de un usuario. Estos scopes pueden variar dependiendo del entorno, es decir que no son iguales en todos los sitios. Lo ideal es que se definan de manera inequívoca. También definden que información se retorna al solicitante del token.

**Tipos de clientes**

OAuth 2.0 reconoce dos tipos de clientes:

* **Clientes confidenciales**: son aquellos que son capaces de guardar una contraseña sin que esta sea expuesta.
* **Clientes públicos**: aquellos que no pueden mantener una contraseña a salvo.

Dependiendo del tipo de cliente que tengamos valoraremos una forma u otra de obtener un token de acceso.

**Diferentes formas de obtener un token de acceso**

Debido a que existen diferentes tipos de aplicaciones y necesidades, existen diferentes formas de obtener un token de acceso. Estas formas en OAuth 2.0 se llaman Flows o Flujos y deberás utilizar una u otra dependiendo del tipo de aplicación cliente que tengas.

**Authorization Code Flow**

Este es el flujo que te he mostrado a través de la imagen, y es el más completo, y por lo tanto el más seguro. Se utiliza con lo que se llaman **confidential clients**, que son aplicaciones que pueden guardar una contraseña (secreto). Este secreto debe ser guardado en un sitio donde no pueda ser accedido a través del cliente. Es decir, **este secreto no se puede guardar en una aplicación hecha en JavaScript, donde el usuario podría navegar a través del código y encontrarlo.** Este escenario aplica perfectamente para websites que tienen un back end seguro.

Entrando más en detalle que en la imagen anterior, esto es lo que ocurre cuando comienza el flujo:

Primero se redirige al usuario al endpoint de autorización, el cual la aplicación cliente conoce, con una serie de parámetros:

https://[authorization.server.com/authorize](http://authorization.server.com/authorize)

?response\_type=code

&client\_id=213f6de8-f232-4854-8c20-80a9b385cca7

&redirect\_uri=<https://client.example.com/callback>

&state=abc

&scope=returngis\_api.read api2.readAndWrite

Según la especificación, estos son los parámetros obligatorios que necesita este tipo de autorización:

* **response\_type**: este parámetro es el que dice qué tipo de flujo de OAuth 2.0 vamos a seguir para recuperar el token, en este caso el valor debe ser **code**.
* **client\_id:**se trata de un identificador de mi aplicación cliente, registrado en el servidor de autorización. Durante la explicación a alto nivel te dije que el servidor de autorización debe verificar que el cliente que solicita el token debe ser un cliente válido. Es por ello que todo cliente que quiera solicitar permisos sobre nuestros recursos protegidos debe de estar registrado en nuestro servidor de autorización. Cuando esto ocurre se asocia un Id a esa aplicación registrada en el servidor que representa a nuestra aplicación cliente.
* **redirect\_uri**: cuando la autenticación del cliente finalice, necesitamos especificar una URL de vuelta a nuestra aplicación. Además, esta URL también está guardada como parte del registro de nuestra aplicación cliente en el servidor de autorización.
* **state**: es recomendado, pero no es obligatorio. Nos permite confirmar que la respuesta que recibimos por parte del servidor es lícita, de tal forma que nos aseguramos que ningún malo ha cambiado la respuesta del servidor por el camino.
* **scope**: se utiliza para decir el “para qué quiero esta autorización”. Son los permisos que se están pidiendo sobre nuestra API. La forma de especificar varios scopes es a través de un espacio en blanco entre ellos.

Cuando el usuario se ha validado correctamente, el servidor de autorización responderá con lo siguiente:

https://[client.example.com/callback](http://client.example.com/callback)

?code=xxxxxxxxxxx

&state=abc

Code representa el consentimiento del usuario y su autorización, y tiene un tiempo de vida bastante corto. Tenemos además el parámetro state que debería de ser igual al que enviamos en la primera petición. Con este código la aplicación hará una llamada a través de un POST al servidor de autorización con el siguiente formato:

POST /token HTTP/1.1

Host: [server.example.com](http://server.example.com/)

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

grant\_type=authorization\_code

&code=xxxxxxxxxxx

&redirect\_uri=<https://client.example.com/callback>

&client\_id=213f6de8-f232-4854-8c20-80a9b385cca7

&client\_secrect=nH7TbHkgsjOWIAtb4NV78RQD5EOtOH16nIusaVzZ4EI=

Se puede utilizar autenticación básica o indicar el client\_id y el client\_secret en el body junto con el resto. Este client\_secret es el secreto del que te hablaba y es obtenido durante el registro de la aplicación en el servidor de autorización.

Si todo va bien, recibiremos la siguiente respuesta:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token" : "una cadena muy larga",

"token\_type" : "Bearer",

"expires\_in" : 3600,

"scope" : "returngis\_api.read api2.readAndWrite"

}

**Implicit Flow**

¿Qué ocurre si la aplicación cliente no puede guardar el secreto? Suele ser en casos donde el usuario puede llegar a ver el client secret, ya que no hay una forma de ocultar el mismo.

Este tipo está diseñado para los que se conocen como **clientes públicos**, que son los que no pueden guardar el secreto. Por ejemplo, un cliente público es una aplicación en AngularJS, que se comunica desde el navegador directamente contra un recurso protegido. De hecho, este flujo apareció específicamente para aplicaciones en JavaScript. Este tipo es menos seguro que el anterior . En este caso el cliente recibe directamente el token y no hay un código intermediario como en el anterior.

La llamada es de esta manera:

https://[authorization.server.com/authorize](http://authorization.server.com/authorize)

?response\_type=token

&client\_id=213f6de8-f232-4854-8c20-80a9b385cca7

&redirect\_uri=<https://client.example.com/callback>

&state=xyz

&scope=returngis\_api.read api2.readAndWrite

En este modo de obtención de los tokens, redirect\_uri es nuestra mayor defensa a la hora de que aplicaciones registradas puedan pedir tokens.

Si todo va bien la respuesta será de esta forma:

https://[client.example.com/callback#](http://client.example.com/callback)

?access\_token=ASFSFd34fsdfde32sdkfsdfjeijfsdlkfjdsklfjdsf...

&token\_type=example

&expires\_in=3600

&state=xyz

Como ves, a diferencia del caso anterior, **el access\_token se expone al usuario final**. Además, cualquier JavaScript en el cliente podría acceder a esta información y usarla. También es posible que una app maliciosa inyectara el token de otro usuario y comenzaras a enviar información personal al usuario erróneo. Es posible usarlo cuando se trata de aplicaciones en las que se confía.

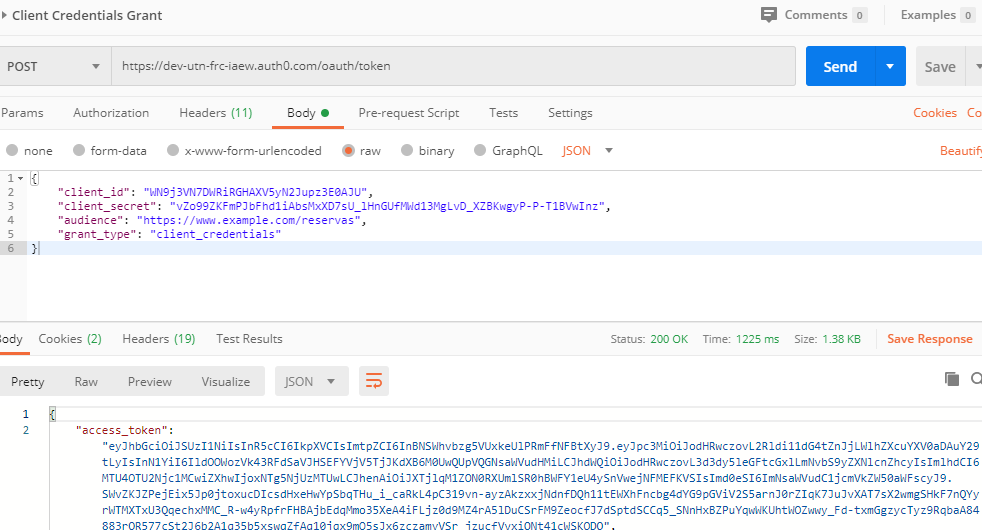
Ejemplo: <https://sso.telecom.com.ar/openam/oauth2/realms/root/realms/tabpinsext/authorize?response_type=id_token%20token&client_id=QshWdye79eGPyySJCewOz5HaO4aSCV28XjXLEAYYDK12Nh87SX0A8q1zFQhD&redirect_uri=https://login.telecom.com.ar/CallbackController.aspx&scope=openid%20profile&state=e74634f2-f58d-4b3e-a27b-573ba26c8c4f&nonce=b3a3a93e-d7c3-405a-b49b-8d2a487c67e0&no_back=true&app=true>

.

**Client Credentials Flow**

¿Y qué ocurre si no hay un usuario propietario de los recursos? ¿y si la aplicación no tiene usuarios involucrados? ¿podemos aún así proteger nuestras APIs con OAuth? La respuesta es sí y es a través de este tipo.

Está pensado donde la aplicación cliente en sí es el resource owner y no hay usuarios involucrados en la operación. Es una comunicación máquina a máquina o Machine to Machine.



|  |
| --- |
| curl --location --request POST 'https://dev-utn-frc-iaew.auth0.com/oauth/token' \  --header 'content-type: application/json' \  --header 'Content-Type: text/plain' \  --header 'Cookie: did=s%3Av0%3A146a73f0-96d6-11ea-a300-8f5addd53441.sZH6l7RS4O21%2BvGI1OOUUSIet0AGtTYq84p90rAOA5M; did\_compat=s%3Av0%3A146a73f0-96d6-11ea-a300-8f5addd53441.sZH6l7RS4O21%2BvGI1OOUUSIet0AGtTYq84p90rAOA5M' \  --data-raw '{  "client\_id": "WN9j3VN7DWRiRGHAXV5yN2Jupz3E0AJU",  "client\_secret": "vZo99ZKFmPJbFhd1iAbsMxXD7sU\_lHnGUfMWd13MgLvD\_XZBKwgyP-P-T1BVwInz",  "audience": "https://www.example.com/reservas",  "grant\_type": "client\_credentials"  }' |

La petición es de la siguiente manera:

POST /token HTTP/1.1

Host: [authorization.server.com](http://authorization.server.com/)

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

grant\_type=client\_credentials

&client\_id=213f6de8-f232-4854-8c20-80a9b385cca7

&client\_secret=nH7TbHkgsjOWIAtb4NV78RQD5EOtOH16nIusaVzZ4EI=

&scope=returngis\_api.read api2.readAndWrite

En respuesta recibimos el siguiente JSON:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token" : "una cadena muy larga",

"token\_type" : "Bearer",

"expires\_in" : 3600,

"scope" : "returngis\_api.read api2.readAndWrite"

}

**Resource Owner Password Credentials (ROPC) Flow**

Está pensado para aplicaciones legacy (Por ejemplo, si tu aplicación usa HTTP Basic Authentication), para que puedan utilizar la arquitectura que provee OAuth 2.0. En este caso mandamos directamente el nombre de usuario y la contraseña para obtener el token. La petición del token en este caso sería de la siguiente manera:

POST /token HTTP/1.1

Host: [authorization.server.com](http://authorization.server.com/)

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

grant\_type=password

&username=gisela

&password=el-password-de-gisela

&client\_id=xxxxxxxxxx

&client\_secret=xxxxxxxxxx

&scope=returngis\_api.read api2.readAndWrite

y la respuesta sería exactamente la misma que la anterior:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token" : "una cadena muy larga",

"token\_type" : "Bearer",

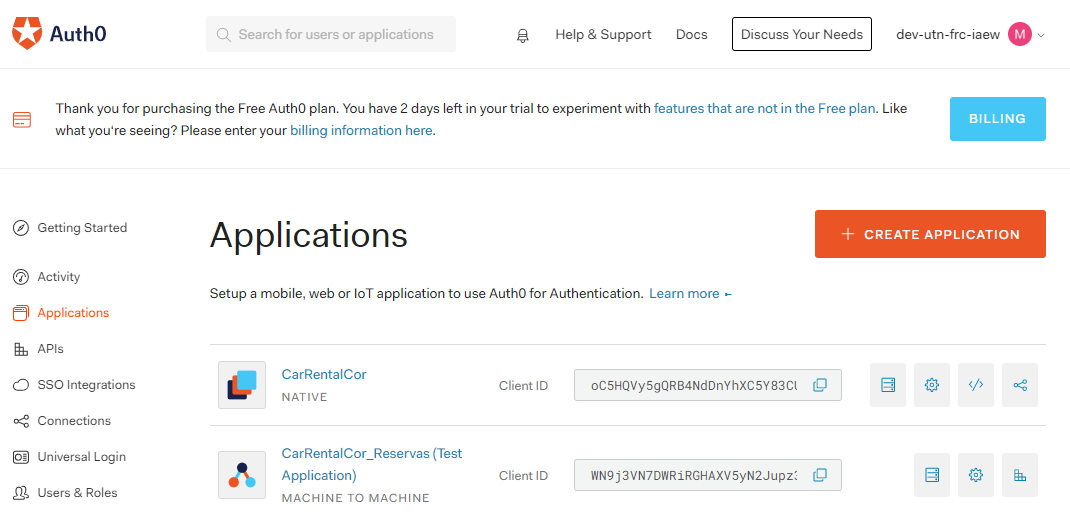
"expires\_in" : 3600,

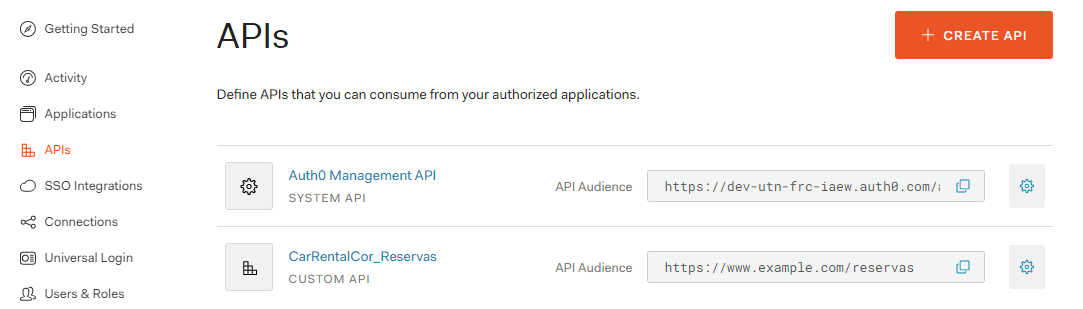
"scope" : "returngis\_api.read api2.readAndWrite"

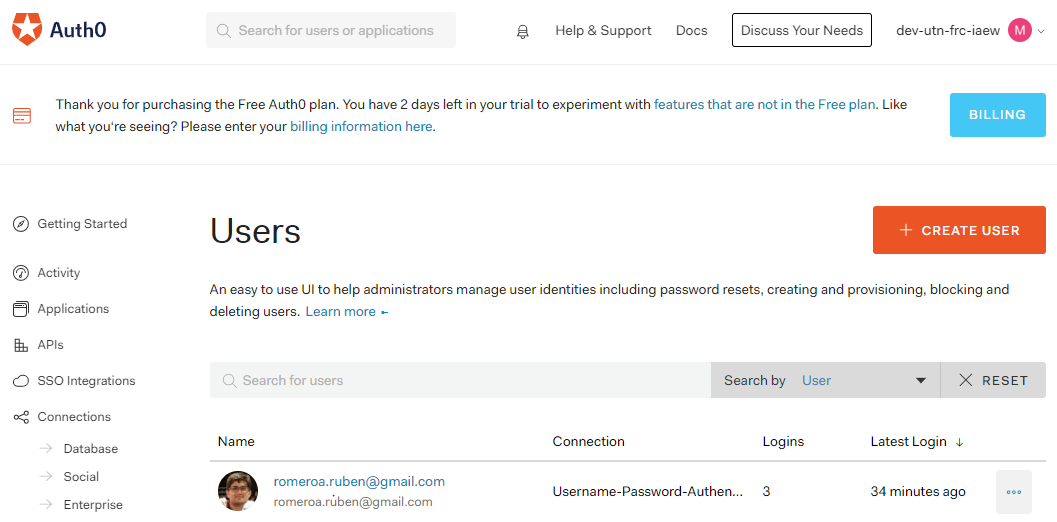
}

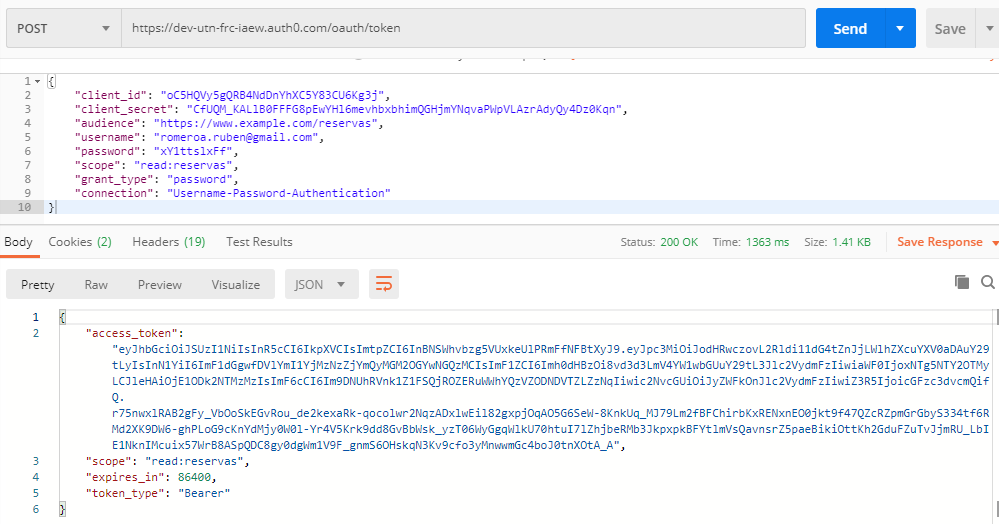
Este tipo tiene las mismas debilidades que compartir el usuario y la contraseña directamente con el recurso protegido.

Ejemplo:









|  |
| --- |
| curl --location --request POST 'https://dev-utn-frc-iaew.auth0.com/oauth/token' \  --header 'content-type: application/json' \  --header 'Content-Type: text/plain' \  --header 'Cookie: did=s%3Av0%3A146a73f0-96d6-11ea-a300-8f5addd53441.sZH6l7RS4O21%2BvGI1OOUUSIet0AGtTYq84p90rAOA5M; did\_compat=s%3Av0%3A146a73f0-96d6-11ea-a300-8f5addd53441.sZH6l7RS4O21%2BvGI1OOUUSIet0AGtTYq84p90rAOA5M' \  --data-raw '{  "client\_id": "oC5HQVy5gQRB4NdDnYhXC5Y83CU6Kg3j",  "client\_secret": "CfUQM\_KALlB0FFFG8pEwYHl6mevhbxbhimQGHjmYNqvaPWpVLAzrAdyQy4Dz0Kqn",  "audience": "https://www.example.com/reservas",  "username": "romeroa.ruben@gmail.com",  "password": "xY1ttslxFf",  "scope": "read:reservas",  "grant\_type": "password",  "connection": "Username-Password-Authentication"  }' |

**Device Code Flow**

Este último tipo se trata de una extensión a OAuth 2.0, debido a nuevas necesidades que han surgido al cabo de los años. Y es que ahora tenemos dispositivos que no tienen un navegador, que nos permita que el usuario pueda acceder al servidor de autorización, como pueden ser el termostato de mi casa, mi cuenta de Youtube en la TV, etcétera.

El flujo que se sigue en este caso es el siguiente:

* El dispositivo en cuestión hace una llamada al servidor de autorización. Este le otorga **un código para que el usuario lo valide**.
* El dispositivo nos pide que accedamos al servidor de autorización **desde otro dispositivo, iniciemos sesión** con nuestras credenciales e introduzcamos el código que le dio.
* El dispositivo va a estar continuamente preguntando al servidor por el token de acceso. Hasta que el usuario no lo valide, este devolverá un error.
* Una vez que nos autentiquemos correctamente, y validemos el código, el servidor le dará el token al dispositivo.

En este caso, la llamada que el dispositivo hace al servidor de autorización tiene un endpoint diferente: **/device\_authorization**

POST /device\_authorization HTTP/1.1

Host: [server.example.com](http://server.example.com/)

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

client\_id=213f6de8-f232-4854-8c20-80a9b385cca7

El servidor devolverá una respuesta parecida a la siguiente, donde se facilita la URL donde el usuario se debe de dirigir para autenticarse y el código que debe introducir para que el dispositivo pueda obtener el token:

{

"device\_code" : "dSFsdfdgdFerEsfDsfdsFsdFerDFdsDASTJYhgFG",

"user\_code" : "WDKF-MSDE",

"verification\_uri" : "<https://example.com/device>",

"verification\_uri\_complete" : "<https://example.com/device?user_code=WDKF-MSDE>",

"expires\_in" : 1800,

"interval" : 5

}

En [este artículo](https://www.returngis.net/2019/04/ejemplo-de-device-code-flow-de-oauth-2-0/) tienes un ejemplo de cómo funciona.

**Ya tienes un token ¿ahora qué?**

Ahora puedes hacer tus peticiones al recurso protegido (API) utilizando tu access token como parte de la cabecera. Normalmente así:

Authorization: Bearer <tu\_access\_token>

**¿Qué ocurre cuando los token expiran?**

En todos los ejemplos hemos visto que **el token que devuelve el servidor tiene un tiempo de expiración**. Dependiendo del escenario en el que estemos trabajando, esto puede no ser viable, ya que hay ocasiones en las que el usuario da acceso a la aplicación y este no espera tener que acceder a ella cada X tiempo a renovar el acceso. Aquí es donde entran en juego los **refresh tokens**. Esto va a suponer que cuando el servidor de autorización nos de el access token, también nos va a dar otro token más que nos permitirá renovar este primero cuando expire. No está soportado por todos los tipos que hemos visto, ya que esto va a significar que si tienes en tu poder un refresh token vas a poder conseguir tokens de acceso directamente. Normalmente se utiliza con clientes confidenciales, de los que son capaces de guardar una contraseña de forma segura.

Para poder pedir este tipo de token hay que agregar como scope el llamado **offline\_access**.

Para poder pedir un nuevo token de acceso, a través del refresh token, hay que hacer la siguiente petición. Esta se hace o bien cuando el tiempo expira o bien cuando el servidor nos devuelve un 401 porque el access token que tenemos ha expirado:

POST /token HTTP/1.1

Host: [server.example.com](http://server.example.com/)

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

grant\_type=refresh\_token

&refresh\_token=dfkjdlksfnlksdjfesjfkslfjkldf

&scope=returngis\_api.read api2.readAndWrite

El refresh token nunca debe ser expuesto al navegador. La respuesta sería como la que sigue:

HTTP/1.1 200 OK

Content-Type: application/json

{

"access\_token" : "una cadena muy larga",

"token\_type" : "Bearer",

"expires\_in" : 3600,

"refresh\_token" : "una cadena muy larga",

"scope" : "returngis\_api.read api2.readAndWrite"

}

Los refresh token suelen ser de un solo uso. Según la especificación, no todos los tipos pueden usar refresh tokens. Solo Authorization Code y ROPC. Implicit y Client Secrets no pueden porque no pueden guardar secretos en el lado del cliente.

Lo habitual es que los clientes de OAuth que utilices en tus aplicaciones gestionen este refresco del token.

**Aplicaciones nativas & OAuth 2.0**

Cuando hablamos de aplicaciones nativas nos referimos a aplicaciones de escritorio y a las aplicaciones móviles.

Estas tienen las siguientes particularidades:

* No hay una URL a la que redirigirse, por lo que no pueden recibir tokens a través del navegador.
* Otras aplicaciones podrían ver nuestro token, por lo que estamos en riesgo.
* No pueden guardar un secreto.

Dicho todo esto, no debería de utilizarse el modo Implicit Flow para este tipo de aplicaciones. Tampoco es recomendable el uso de navegadores embebidos dentro de la aplicación. Para este caso tenemos una técnica llamada **Proof Key for Code Exchange (PKCE)**, que funciona de la siguiente manera:

* Antes de que el cliente comience el proceso de autorización, este genera un valor aleatorio llamado **code\_verifier**.
* En la primera llamada al servidor de autorización se incluye un hash de este valor. Este hash es llamado el **code\_challenge**.
* El servidor guardará este code\_challenge.
* El proceso sigue igual, si la validación es correcta el servidor devolverá el código con el cuál pedir el token.
* A la hora de pedir el token el cliente envía el code\_verifier, no el hash.
* El servidor compara el code\_verifier con el hash que el tiene, para ver si son el mismo.
* Si estos son iguales el servidor puede confirmar que quien está pidiendo el token es el mismo que mandó el código al principio, y no una aplicación maliciosa que robó el código, por lo que mandará el access token.

En lugar de utilizar un navegador embebido en tu aplicación, haz uso del navegador del sistema, el cual tiene integración con el gestor de contraseñas y permite single sign-on. Por ejemplo, esto es lo que usa Facebook por ejemplo. [Aquí](https://oauth.net/2/native-apps/) puedes encontrar más información.

**OpenID Connect**

Después de haber dado un repaso a OAuth y los diferentes flujos que tenemos para obtener un token, podemos ver que este no es perfecto y que tiene carencias frente a algunas necesidades. Por ejemplo:

* Solo es un framework de autorización
* No es capaz de identificar a los usuarios

OpenID Connect está pensado para la autenticación y OAuth para la autorización. Este añade las siguientes funcionalidades que complementan a OAuth:

* Un ID token que nos permite saber quién es el usuario.
* Un nuevo endpoint, **UserInfo**, que nos permite recuperar más información del usuario.
* Un conjunto de scopes estándar.
* Un conjunto de claims que nos permite obtener datos del sujeto.

**ID token**

En el momento que añades OpenID Connect a tu flujo de OAuth 2.0, la respuesta obtenida será parecida a la siguiente:

{

"access\_token": "fFAGRNJru1FTz70BzhT3Zg",

"id\_token": "eyJraBsdsdw3F..."

"expires\_in": 3920,

"token\_type": "Bearer",

}

Como puedes ver, además del access token que veníamos recuperando con OAuth 2.0, gracias a OpenID Connect tenemos un nuevo token llamado id token. Este nos va a proporcionar información sobre el usuario del cual estamos recibiendo la autorización. Esta información viene en formato J**WT o JSON Web Token, el cual es un estándar (**[**https://tools.ietf.org/html/rfc7519**](https://tools.ietf.org/html/rfc7519)**)**, que pretende transmitir de manera segura información entre dos partes en formato JSON. Esta información puede ser verificada, ya que normalmente está digitalmente firmada.  
  
La estructura de los tokens en este formato consiste en tres partes separadas por puntos, algo parecido a esto: **xxxxxxx.yyyyy.zzzzz**. Cada parte significa lo siguiente:

* La **cabecera**, la cual normalmente tiene dos partes: el tipo de token y el algoritmo que se ha utilizado para ser firmado. Este puede ser HMAC SHA256 o RSA.
* El **payload**, esta es la parte que contiene los claims, que en español podríamos traducirlo como los *derechos,* que tiene un usuario. También suele haber otro tipos de claims adicionales. Este payload está en Base64 y forma la segunda parte del token
* La **firma**: esta parte se utiliza para verificar que el mensaje no ha sido modificado durante el envío y, en el caso de los tokens que han sido firmados con una clave privada, además puede verificar que el emisor del token es quien dice ser.

El resultado de todo esto son tres cadenas codificadas en Base64-URL que pueden ser fácilmente enviadas a través de un entorno web y es mucho más compacta, comparada con el XML del que veníamos. En el caso del ID token sería algo así:

(Header)

.

{

"iss": "[https://accounts.returngis.net](https://accounts.returngis.net/)",

"sub": "you@returngis.net",

"name": "Gisela Torres"

"aud": "s7HktUspc3",

"exp": 1311280970,

"iat": 1311281970,

"auth\_time": 1311280969,

}

.

(Signature)

Existen diferentes páginas para decodificar y validar este tipo de tokens. Quizás la más famosa es [jwt.io](https://jwt.io/), de Auth0, pero también hay otras:

* <https://www.jsonwebtoken.io/>
* <https://jwt.ms/>

**UserInfo**

Una vez que tienes un ID token puedes usar este mismo para recuperar más información acerca del usuario:

HTTP/1.1 200 OK

Content-Type: application/json

{

"sub": "248289761001",

"name": "Jane Doe",

"given\_name": "Jane",

"family\_name": "Doe",

"preferred\_username": "j.doe",

"email": "janedoe@example.com",

"picture": "<http://example.com/janedoe/me.jpg>"

}

**Conjunto de scopes estándar**

OpenID Connect define una serie de scopes estándar, que son los siguientes:

* **openid**: si quieres usar OpenID Connect este es requerido.
* **profile**: pide los siguientes claims del usuario: name, family\_name, given\_name, middle\_name, nickname, preferred\_username, profile, picture, website, gender, birthdate, zoneinfo, locale, y updated\_at.
* **email**: pide el acceso al email y al email verificado del usuario.
* **address**: recupera el claim que contiene la dirección del usuario.
* **phone**: recupera el teléfono y el teléfono verificado del usuario.

Al igual que existen un conjunto de scopes estándar, también existe un [conjunto de claims estándar.](https://openid.net/specs/openid-connect-core-1_0.html#StandardClaims)

**Hybrid Flow**

Con OpenID Connect aparece además un nuevo flujo llamado **Hybrid Flow**. Se trata de una **combinación entre el Authorization Code Flow y el Implicit Flow**. Dependiendo del response\_type que solicites, podrás recuperar algunos tokens desde el authorization endpoint y otros desde el token endpoint. Este tipo se utiliza cuando quieres que el usuario tenga acceso inmediato al token de identidad (id\_token), pero también quieres hacer llamadas al back end para intercambiar un código de autorización por un token de acceso. El tipo de respuesta debe tener de forma obligatoria el tipo code y cualquiera de las siguientes combinaciones:

* response\_type=code token
* response\_type=code id\_token
* response\_type=code id\_token token

**Resumen**

OAuth 2.0 es para autorización y OpenID Connect es para autenticación de usuario final.

**¿Qué flujo debería de seguir?**

* **Aplicaciones con back end**: Authorization Code Flow.
* **Aplicaciones web sin back end**: Implicit Flow o Hybrid Flow.
* **Aplicaciones nativas**: Authorization Code flow con PCKE (https://tools.ietf.org/html/rfc7636).
* **Aplicaciones sin usuarios**: Client Credentials Flow.
* **Dispositivos sin navegador**: OAuth Device Flow