

Programación Web 3

UNLaM - Tecnicatura en Desarrollo Web

Trabajo Práctico de Investigación

Login – Autenticación Integrada

Integrantes	1
Objetivo	2
Situación Actual	2
Desarrollo de la Investigación	3
Conclusiones	18
Referencias bibliográficas	18

Integrantes

Barone, Esteban Nicolas

Martínez, Laura Giselle

Navarro, Francisco Javier

Persico, Camila Belén

Wagner, Ian Lautaro

Objetivo

El siguiente trabajo de investigación focalizará en el tema del login y de la autenticación integrada con el objetivo de analizar los distintos tipos de autenticación en .NET y comparar sus diferencias e implementaciones.

Situación Actual

En primer lugar, OpenID Connect u OIDC es un protocolo de identidad que utiliza los mecanismos de autorización y autenticación de OAuth 2.0. La especificación final de OIDC se publicó el 26 de febrero de 2014 y, actualmente, es ampliamente adoptada por muchos proveedores de identidad en Internet. Mientras que OAuth 2.0 es un protocolo de autorización, OIDC es un protocolo de autenticación de identidad y puede utilizarse para verificar la identidad de un usuario ante un servicio de cliente.

Las especificaciones de OpenID son desarrolladas por grupos de trabajo de la *Fundación OpenID* (que incluye empresas como Google y Microsoft) en que se dividen en tres fases diferentes: borradores, borradores del implementador y especificaciones finales. Los borradores del implementador y las especificaciones finales brindan protecciones de propiedad intelectual a los implementadores. Las especificaciones finales son los estándares de OpenID Foundation.

En segundo lugar, OAuth 2.0, que significa “Open Authorization” (autorización abierta), es un estándar diseñado para permitir que un sitio web o una aplicación accedan a recursos alojados por otras aplicaciones web en nombre de un usuario. Sustituyó a OAuth 1.0 en 2012 y ahora es el estándar de facto de la industria para la autorización en línea. OAuth 2.0 proporciona acceso consentido y restringe las acciones que la aplicación del cliente puede realizar en los recursos en nombre del usuario, sin compartir nunca las credenciales del usuario.

Aunque la web es la principal plataforma para OAuth 2, la especificación también describe cómo manejar este tipo de acceso delegado a otros tipos de clientes (aplicaciones basadas en el navegador, aplicaciones web del lado del servidor, aplicaciones nativas/móviles, dispositivos conectados, etc.).

Desarrollo de la Investigación

OpenID Connect

Es un protocolo de identidad que utiliza los mecanismos de autorización y autenticación de OAuth 2.0.

OpenID Connect, al estar pensado para la autenticación, 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. Cuando se añade OpenID Connect al flujo de OAuth 2.0, la respuesta obtenida será semejante a la siguiente:

```
{
  "access_token": "fFAGRNJru1FTz70BzhT3Zg",
  "id_token": "eyJraBsdsw3F..."
  "expires_in": 3920,
  "token_type": "Bearer",
}
```

Además del access token que se recuperaba con OAuth 2.0, gracias a OpenID Connect aparece un nuevo token llamado *id token*. Este brinda información sobre el usuario del cual estamos recibiendo la autorización. Esta información viene en formato JWT o JSON Web Token. El JWT es un estándar que busca 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 esta estructura: xxxxxxxx.yyyyyy.zzzzz. Veamos qué significa cada parte:

- La *cabecera o header* 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* contiene los claims, es decir, los *derechos* que tiene un usuario. También suele haber otros tipos de claims adicionales. Podemos diferenciar entre *registered* (predefinidos y recomendados por el estándar), *public* (definidos a voluntad y sin

restricciones aunque con recomendaciones) y *private* (completamente personalizados para compartir información acordada entre las partes de manera concreta).

- La *firma o signature* 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. (Incluye la llave secreta para validar el token)

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 antiguo. En el caso del ID token sería algo así:

(Header)

```
.
{
  "iss": "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. La más famosa es jwt.io, de Auth0, pero también hay otras:

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

UserInfo. Una vez obtenido un ID token se puede 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:

- *openid*: si quieres usar OpenID Connect este es requerido.
- *profile*: pide los claims del usuario tales como `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.

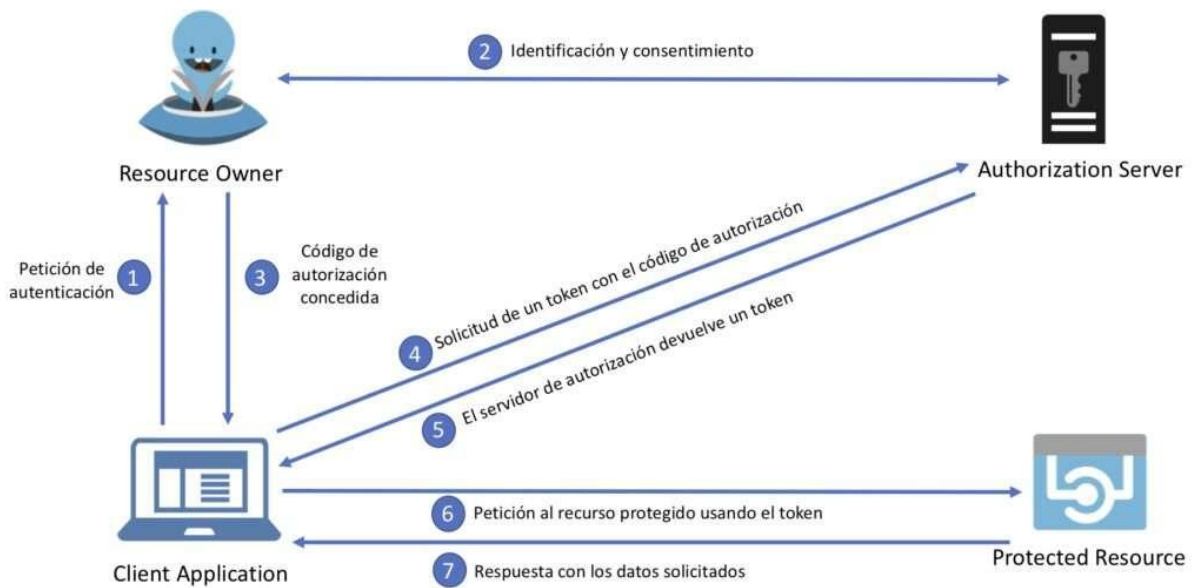
Claims. Son un conjunto de reclamaciones estándar que se encargan de pedir mayor información del usuario. (Por ejemplo, nombre completo del usuario final, fecha de nacimiento, género, etc).

Proceso de autorización

Una vez finalizado el proceso de autenticación mediante el cual se verifica la identidad de un usuario, se debe implementar mecanismos de autorización para controlar qué permisos tiene cada miembro. En este sentido, *OAuth* es un framework que se construyó específicamente para acceder a APIs a través de HTTP. En OAuth 2.0 intervienen 4 partes:

- **Protected Resource:** El recurso al que queremos acceder, una API.
- **Cliente:** 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 suya los datos que maneja sí lo son.
- **Authorization server:** es el responsable de gestionar las peticiones de autorización.

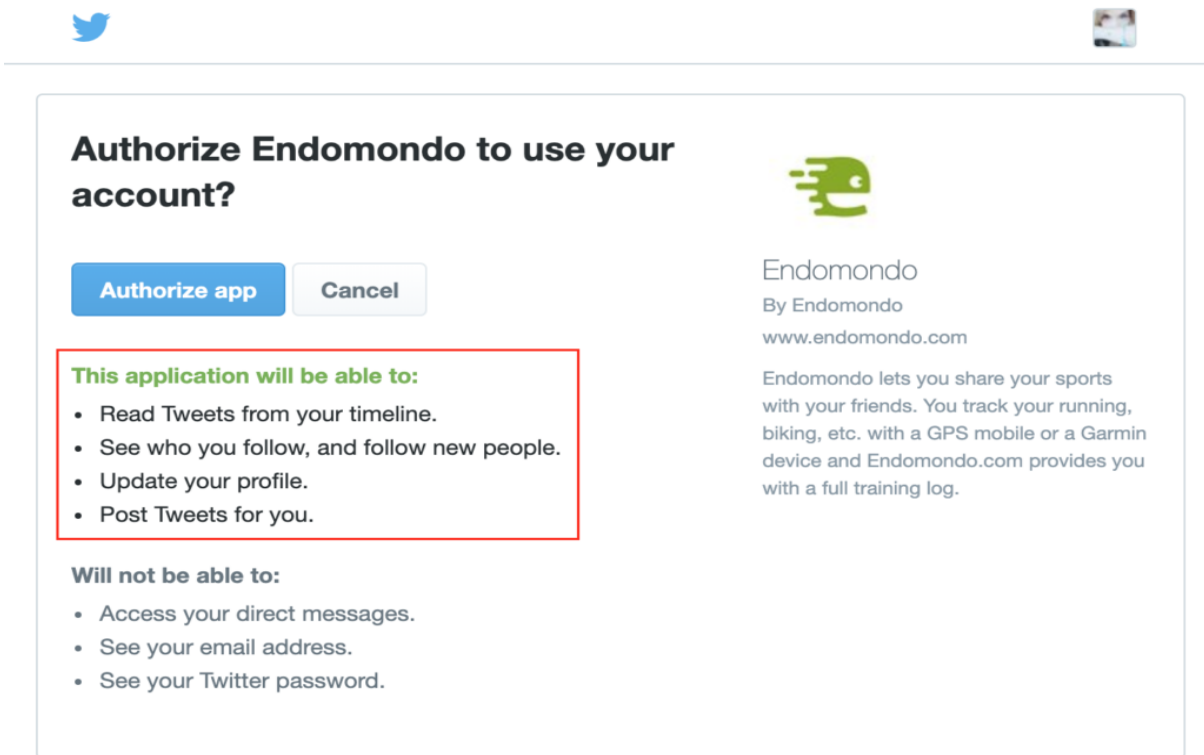
La forma en la que estas 4 partes se relacionan entre sí es la siguiente:



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, de reconocimiento facial, de voz, con 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, etc. 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. Luego, 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.
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 es así, la API responderá con los datos que se le han pedido.

Ejemplo de consentimiento (Consent)



Twitter pide el consentimiento del usuario para otorgarle a Endomondo permisos

Endpoints. Para los pasos 2 y 4, en OAuth hay dos URLs en el 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.

Scopes. Los scopes son los permisos para hacer algo dentro de un recurso protegido en nombre de un usuario. Estos pueden variar dependiendo del entorno, es decir que no son iguales en todos los sitios. Lo ideal es que se definan de manera inequívoca. Es decir, en lugar del scope read que sea un scope `returngis_api.read`, ya que refleja claramente que este es solo el acceso en modo lectura a una API en concreto.

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.

Flujos en OAuth 2.0

Debido a que existen diferentes tipos de aplicaciones y necesidades, existen diferentes formas de obtener un token de acceso.

Authorization Code Flow. Este es el flujo 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.

Cuando comienza el flujo, primero se redirige al usuario al endpoint de autorización con una serie de parámetros, que la aplicación cliente conoce:

`https://authorization.server.com/authorize`

`?response_type=code`

`&client_id=213f6de8-f232-4854-8c20-80a9b385cca7`

`&redirect_uri=https://client.example.com/callback`

`&state=abc`

`&scope=returnis_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 la aplicación cliente, registrado en el servidor de autorización. El servidor de autorización debe verificar que el cliente que solicita el token debe ser un cliente válido, por ello 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*: URL guardada como parte del registro de la aplicación cliente en el servidor de autorización.
- *state*: es recomendado, pero no es obligatorio. Permite confirmar que la respuesta que recibimos por parte del servidor es lícita y que no ha sido manipulada.
- *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 así:

`https://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

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_secret=nH7TbHkgsjOWIAtb4NV78RQD5EOtOH16nIusaVzZ4EI=`

Se puede utilizar autenticación básica o meter el `client_id` y el `client_secret` en el body junto con el resto. Este `client_secret` es el secreto que la aplicación debe proteger, 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. Este flujo ya no es el recomendado, sino Authorization Code Flow con PKCE.

¿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 flujo fue 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, ya que 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`

`?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#`

`?access_token=ASFd34fsdfde32sdksdfjeijfsdlkfjdsklfjdsf...`

`&token_type=example`

`&expires_in=3600`

`&state=xyz`

Como se ve, 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.

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 2.0? La respuesta es sí y es a través de este flujo.

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.

La petición es de la siguiente manera:

POST /token HTTP/1.1

Host: 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 una aplicación usa HTTP Basic Authentication), para que puedan utilizar la arquitectura que provee OAuth 2.0. Debería de ser una solución temporal, para dar tiempo a cambiar el tipo de autenticación de la aplicación. En este caso mandamos directamente el nombre de usuario y la contraseña para obtener el token. En aplicaciones modernas este tipo no debe ser usado, ya que se considera deprecado.

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

Device Code Flow. Se trata de una extensión a OAuth 2.0, debido a nuevas necesidades que han surgido. En efecto, tenemos dispositivos que no tienen un navegador que nos permita que el usuario pueda acceder al servidor de autorización.

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 autenticemos 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

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" : "dSFsdfgdgDFerEsfDsfdFsdFerDFdsDASTJYhgFG",
  "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
```

```
}
```

Ahora se podría hacer peticiones al recurso protegido (API) utilizando el access token como parte de la cabecera. Normalmente así: Authorization: Bearer <tu_access_token>

¿Qué ocurre cuando los token expiran?

El token que devuelve el servidor tiene un tiempo de expiración. Dependiendo de cada escenario, 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. Como solución, tenemos los *refresh tokens*. Esto va a suponer que cuando el servidor de autorización nos dé el access token, también nos va a dar otro token más que nos permitirá renovar al primero cuando expire. No está soportado por todos los tipos vistos. Normalmente se utiliza con clientes confidenciales, 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 cuando el tiempo expira o cuando el servidor nos devuelve un 401 porque el access token que tenemos ha expirado:

```
POST /token HTTP/1.1
```

```
Host: server.example.com
```

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

```
grant_type=refresh_token
```

```
&refresh_token=dfkjdlksfnlksdjfesjfklsfjkldf
```

```
&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.

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 existe 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 cual 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 él tiene. 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.

¿Qué flujo debería de seguir?

- Aplicaciones con back end: Authorization Code Flow
- Aplicaciones web sin back end: Authorization Code Flow con PKCE.
- Aplicaciones nativas: Authorization Code flow con PKCE
- Aplicaciones sin usuarios: Client Credentials Flow
- Dispositivos sin navegador: OAuth Device Flow

Ventajas de OpenID Connect y OAuth 2.0

- Es fácil, confiable y seguro.
- Elimina el almacenamiento y la administración de las contraseñas de las personas.
- Mejora la experiencia del usuario de alta y registro y reduce el abandono del sitio web.
- Los marcos de autenticación basados en el cifrado de clave pública, como OpenID Connect, aumentan la seguridad de todo Internet al poner la responsabilidad de la verificación de la identidad del usuario en manos de los proveedores de servicios de identidades más expertos.
- OAuth 2.0 permite que el usuario sea consciente de para qué la aplicación solicita su permiso y qué puede hacer con él.
- Supera dos falencias de las ApiKeys al no dar acceso total a todas las operaciones (ya que sigue las especificaciones de autorización de Auth) y al permitir identificar usuarios y ya no proyectos.
- Supera el problema de la suplantación de la identidad, ya que la aplicación no guarda la identificación del usuario.

AZURE ACTIVE DIRECTORY

Azure Active Directory (Azure AD), parte de Microsoft Entity, es un servicio de identidad empresarial que ofrece inicio de sesión único, autenticación multifactor y acceso condicional para proteger contra el 99,9 por ciento de los ataques de ciberseguridad.

La autenticación por Azure Active Directory se basa en comprobar el nombre de usuario, contraseña y para mejorar la seguridad y reducir la necesidad de asistencia del departamento de soporte técnico incluye:

- *Restablecimiento de la contraseña de autoservicio*, sin necesidad de que intervenga un administrador. Si Azure bloquea la cuenta de un usuario o este se ha olvidado su contraseña, puede desbloquearla el mismo usuario. Esto reduce la pérdida de productividad cuando un usuario no puede iniciar sesión en su dispositivo o en una

aplicación. También, puede actualizar o cambiar sus contraseñas mediante un explorador web desde cualquier dispositivo.

Funciona en los siguientes escenarios:

- o Cambio de contraseña: el usuario conoce la contraseña pero desea cambiarla por una nueva.
- o Restablecimiento de contraseña: el usuario no puede iniciar sesión, por ejemplo, ha olvidado la contraseña y quiere restablecerla.
- o Desbloqueo de cuenta: el usuario no puede iniciar sesión porque su cuenta está bloqueada y quiere desbloquearla.

Cuando un usuario actualiza o restablece su contraseña, esta puede reescribirse en un entorno de Active Directory local. La escritura diferida de contraseñas garantiza que un usuario puede usar inmediatamente sus credenciales actualizadas en aplicaciones y dispositivos locales.

- Azure AD Multi-Factor Authentication es un proceso por el que, durante un evento de inicio de sesión, se solicitan a un usuario otras formas de identificación. Por ejemplo, se le puede pedir que introduzca un código en el teléfono móvil, escaneando la huella digital, mediante una llamada telefónica o por una notificación en la aplicación móvil. Al exigir una segunda forma de autenticación, aumenta la seguridad, porque este factor adicional no es fácil de obtener o duplicar para un atacante. Esto reduce el requisito de una única forma fija de autenticación secundaria, como un token de hardware. Si el usuario no dispone actualmente de una forma de autenticación adicional, puede elegir un método diferente y seguir trabajando.
- Protección con contraseña, de forma predeterminada, bloquea las contraseñas no seguras. Se actualiza y se aplica automáticamente una lista global de contraseñas no seguras conocidas. Si un usuario intenta utilizar como contraseña una de estas contraseñas no seguras, recibirá una notificación para elegir otra más segura.
- Autenticación sin contraseñas como Windows Hello, las claves de seguridad FIDO2 y la aplicación Microsoft Authenticator, responder a una notificación push, especificar un código de un token de software o hardware, o responder a un SMS o a una llamada de teléfono, ya que proporcionan un inicio de sesión más seguro.

Autenticación de aplicaciones .NET en servicios de Azure durante el desarrollo local mediante cuentas de desarrollador

Al crear aplicaciones en la nube, los desarrolladores necesitan depurar y probar aplicaciones en su estación de trabajo local. Cuando se ejecuta una aplicación en la estación de trabajo de un desarrollador durante el desarrollo local, esta debe autenticarse.

Para que una aplicación se autentique en Azure durante el desarrollo local mediante las credenciales de Azure del desarrollador, el desarrollador debe iniciar sesión en Azure desde

la extensión VS Code Azure Tools, la CLI de Azure o Azure PowerShell. El SDK de Azure para .NET puede detectar que el desarrollador inició sesión desde una de estas herramientas y, a continuación, obtener las credenciales necesarias de la memoria caché de credenciales para autenticar la aplicación en Azure como usuario que inició sesión.

Para autenticarse con Azure se deben seguir estos pasos:

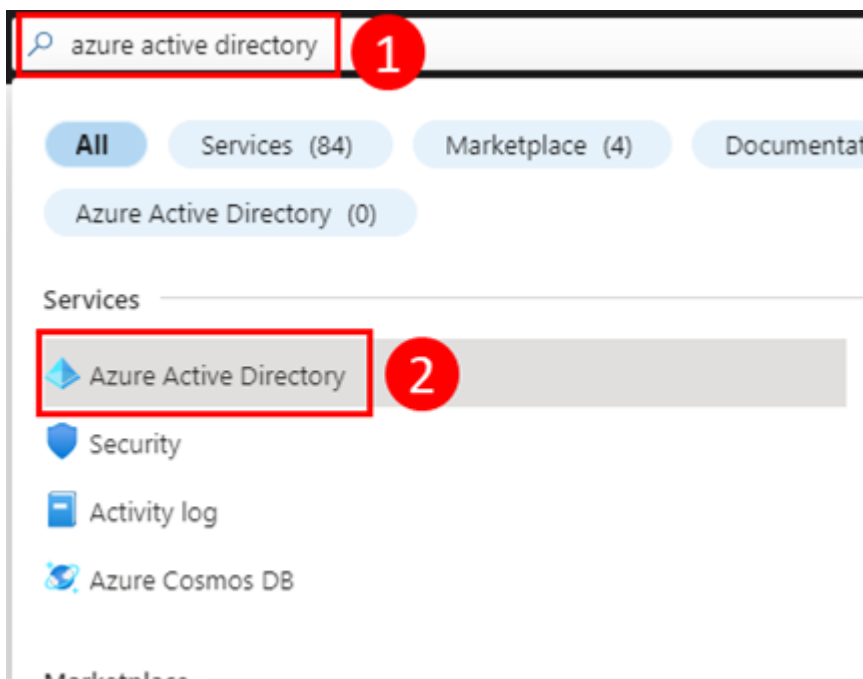
1. Creación de un grupo de Azure AD para el desarrollo local para encapsular los roles (permisos) que la aplicación necesita en el desarrollo local. Esto ofrece las siguientes ventajas:

- a. Todos los desarrolladores están seguros de tener asignados los mismos roles, ya que los roles se asignan en el nivel de grupo.

- b. Si se necesita un nuevo rol para la aplicación, solo debe agregarse al grupo de Azure AD para la aplicación.

- c. Si un nuevo desarrollador se une al equipo, simplemente debe agregarse al grupo correcto de Azure AD para obtener los permisos correctos para poder trabajar en la aplicación.

Para crear el grupo en la página de Inicio - Microsoft Azure buscamos Azure Active Directory, seleccionamos grupos y nuevo grupo. Dentro de este ultimo podemos elegir el tipo de seguridad, el nombre del grupo o de la aplicación que estamos realizando, la descripción y los miembros.



[Home](#) >



Default Directory |

Azure Active Directory



Overview



Preview features



Diagnose and solve problems

Manage



Users



Groups



External Identities


[Home](#) > [Default Directory](#) >

Groups | All groups ...


Default Directory - Azure Active Directory

<< **All groups**


 Deleted groups

 Diagnose and solve problems


Settings


 General


 Expiration


 Naming policy

Activity


 Privileged access groups (Preview)

 Access reviews


 **New group**

 Download groups

 Delete

 Refresh



 Search

Search mode  Contains

1 group found

☐ Name ↑

Obj

☐ **PV** Product Viewers

f65

[Home](#) > [Default Directory](#) > [Groups](#) >

New Group ...

Group type * ⓘ

Security

Group name * ⓘ

msdocs-dotnet-sdk-auth-local-dev

Group description ⓘ

Group defending developers and roles needed for local development of sample app

Membership type ⓘ

Assigned

Owners

No owners selected

Members

No members selected

Create

2. Asignación de roles (permisos) necesita la aplicación, que se pueden asignar a un rol en el ámbito de recurso, grupo de recursos o suscripción.

Se busca el grupo de recursos y se selecciona el control de acceso. Dentro de este último elegimos asignación de roles y agregar, donde se pueden asignar los roles y especificar a que usuarios se le asignan.

Home >

msdocs-dotnet-sdk-auth-example ...

Resource group

Search (Ctrl+/) << + Create Manage view Delete resource group Refresh Export to CSV Open query

Overview

Activity log

Access control (IAM)

Tags

Resource visualizer

Events

Settings

Deployments

Security

Policies

Properties

Locks

Essentials

Subscription (move) : [Pay-As-You-Go](#)

Subscription ID : e322fe5a-eea7-41b2-a2a4-97732b067738

Tags (edit) : [Click here to add tags](#)

Resources Recommendations

Filter for any field... Type == all Location == all Add filter

Showing 1 to 2 of 2 records. Show hidden types

☐ Name ↑↓

☐ msdocs-dotnet-sdk-auth-example

☐ msdocs-dotnet-sdk-auth-example-app-plan

Home > msdocs-dotnet-sdk-auth-example

msdocs-dotnet-sdk-auth-example | Access control (IAM) ...

Resource group

Search (Ctrl+/) << + Add Download role assignments Edit columns Refresh Remove Got feedback?

Overview

Activity log

Access control (IAM)

Tags

Resource visualizer

Events

Check access **Role assignments** Roles Deny assignments Classic administrators


Number of role assignments for this subscription

7 2000

Search by name or email Type : All Role : All Scope : All scopes Group by : Role

[Home](#) > [msdocs-dotnet-sdk-auth-example](#) >

Add role assignment ...

 Got feedback?

[Role](#) [Members](#) [Review + assign](#)

A role definition is a collection of permissions. You can use the built-in roles or you can create your own custom roles. [Learn more](#)

Type : All

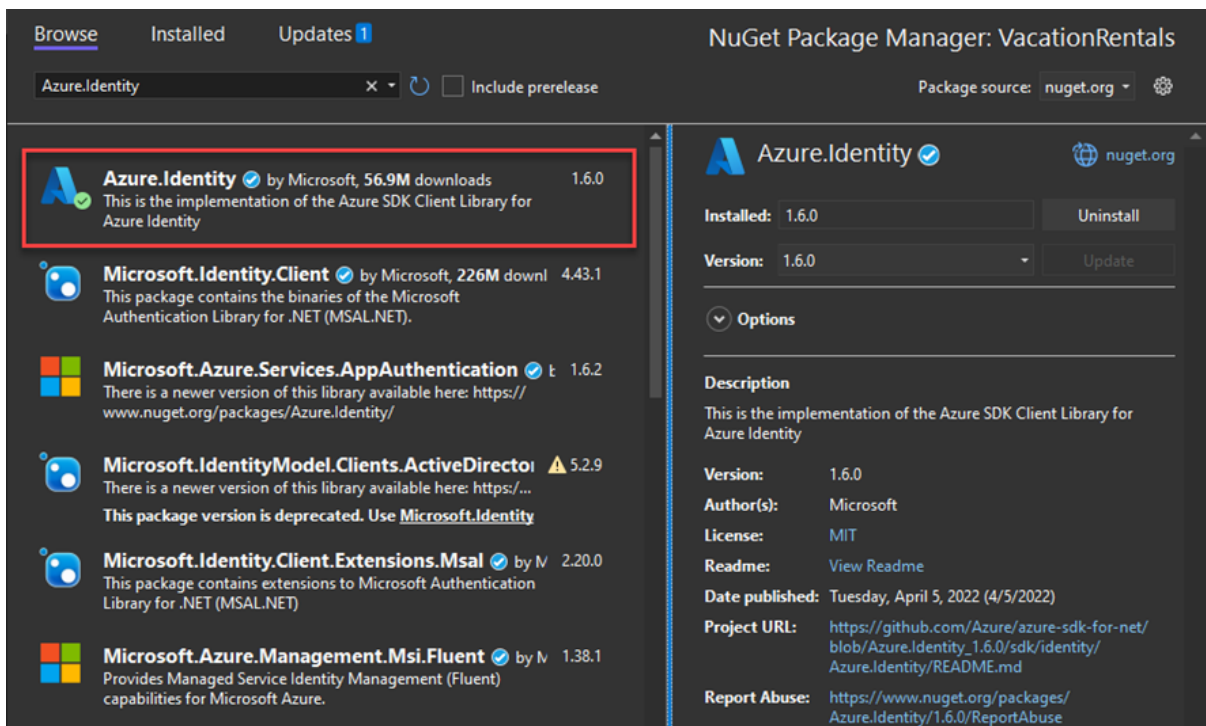
Category : All

Showing 4 of 334 roles

Name ↑↓	Description ↑↓
Storage Blob Data Contributor	Allows for read, write and delete access to Azure Storage blob containers and data
Storage Blob Data Owner	Allows for full access to Azure Storage blob containers and data, including assigning POSIX access control.
Storage Blob Data Reader	Allows for read access to Azure Storage blob containers and data
Storage Blob Delegator	Allows for generation of a user delegation key which can be used to sign SAS tokens

3. Inicio de sesión en Visual Studio, en el menú Herramientas seleccionamos Opciones y en su barra de búsqueda escribimos Azure. En Autenticación de servicio de Azure, elegimos Selección de cuenta y agregamos una cuenta Microsoft.

4. Implementación de DefaultAzureCredential en la aplicación, el cual admite varios métodos de autenticación y determina el método de autenticación que se usa en tiempo de ejecución. De esta manera, la aplicación puede usar diferentes métodos de autenticación en distintos entornos sin implementar código específico del entorno. Para implementar esto se agrega primero Azure.Identity y, opcionalmente, los paquetes Microsoft.Extensions.Azure a la aplicación, mediante el Administrador de paquetes de NuGet.



Por lo general, se accede a los servicios de Azure mediante las clases de cliente correspondientes desde el SDK. Estas clases y sus propios servicios personalizados deben registrarse en el archivo Program.cs para que se pueda acceder a ellas a través de la inserción de dependencias en toda la aplicación.

using Microsoft.Extensions.Azure;

using Azure.Identity;

// Dentro de Program.cs

builder.Services.AddAzureClients(x =>

{

x.AddBlobServiceClient(new Uri("https://<account-name>.blob.core.windows.net"));

x.UseCredential(new DefaultAzureCredential());

});

Como alternativa, también se puede usar `DefaultAzureCredential` en los servicios de manera más directa sin la ayuda de los métodos de registro de Azure adicionales.

```
using Azure.Identity;
```

```
// Dentro de Program.cs
```

```
builder.Services.AddSingleton<BlobServiceClient>(x =>
```

```
    new BlobServiceClient(
```

```
        new Uri("https://<account-name>.blob.core.windows.net"),
```

```
        new DefaultAzureCredential()));
```


Conclusiones

En conclusión, **OAuth 2.0** es un estándar de seguridad ampliamente utilizado y eficaz para la autenticación y autorización de usuarios. Básicamente, permite que la contraseña y el usuario ya no se guarden en el cliente pudiendo ser vulnerados mediante ataques por *cross-site scripting*, entre otros, puesto que delega el servicio de identidad en un tercero. En este trabajo analizamos cómo los tokens intervienen en el proceso de autorización pero para poder autenticar la identidad de un usuario se necesitó complementar esta tecnología con *OpenID Connect*.

Este protocolo de identidad que utiliza los mecanismos de autorización y autenticación de OAuth 2.0 implica numerosos beneficios en la actualidad: elimina el almacenamiento y la administración de las contraseñas de las personas, mejora la experiencia del usuario en el proceso de inicio y de registro de sesión, aumenta la seguridad de todo Internet al apelar a los proveedores de servicios de identidades, permite que el usuario sea consciente de para qué la aplicación solicita su permiso y qué puede hacer con él, supera falencias de las ApiKeys y supera el problema de la suplantación de la identidad, ya que la aplicación no guarda la identificación del usuario.

En efecto, *OAuth2.0* junto a *OpenID Connect*, permite a los usuarios compartir información de manera segura sin tener que compartir sus contraseñas, y ofrece flexibilidad y escalabilidad para los desarrolladores.

Finalmente, *Azure Active Directory* (Azure AD) es un servicio de identidad empresarial que sigue las especificaciones de OAuth2.0 y de OpenID Connect para ofrecer inicio de sesión único, autenticación multifactor y acceso condicional seguros y capaces de proteger contra los ataques de ciberseguridad.

Referencias/Bibliografía

JWT:

- <https://rafaelneto.dev/blog/autenticacion-autorizacion-jwt-bearer-aspnet-core/>
- <https://www.enmilocalfunciona.io/construyendo-una-web-api-rest-segura-con-json-web-token-en-net-parte-ii/>

OPEN ID y OAUTH 2.0:

- <https://auth0.com/es/intro-to-iam/what-is-openid-connect-oidc>
- <https://openid.net/connect/>
- <https://www.returngis.net/2019/04/oauth-2-0-openid-connect-y-json-web-tokens-jwt-que-es-que/>

AZURE

- <https://learn.microsoft.com/es-es/azure/active-directory/authentication/overview-authentication>
- <https://learn.microsoft.com/es-es/dotnet/azure/sdk/authentication-local-development-dev-accounts?tabs=azure-portal%2Csign-in-visual-studio%2Cnuget-package>