EDICIÓN #003

OAuth 2.0 y OpenID Connect (OIDC): la guía definitiva para entender autorización y autenticación

12 de Enero de 2026

OAuth 2.0 y OpenID Connect (OIDC): la guía definitiva para entender autorización y autenticación

Te explico cómo funcionan OAuth 2.0 y OIDC y cómo protegen tus datos sin usar usuario/contraseña

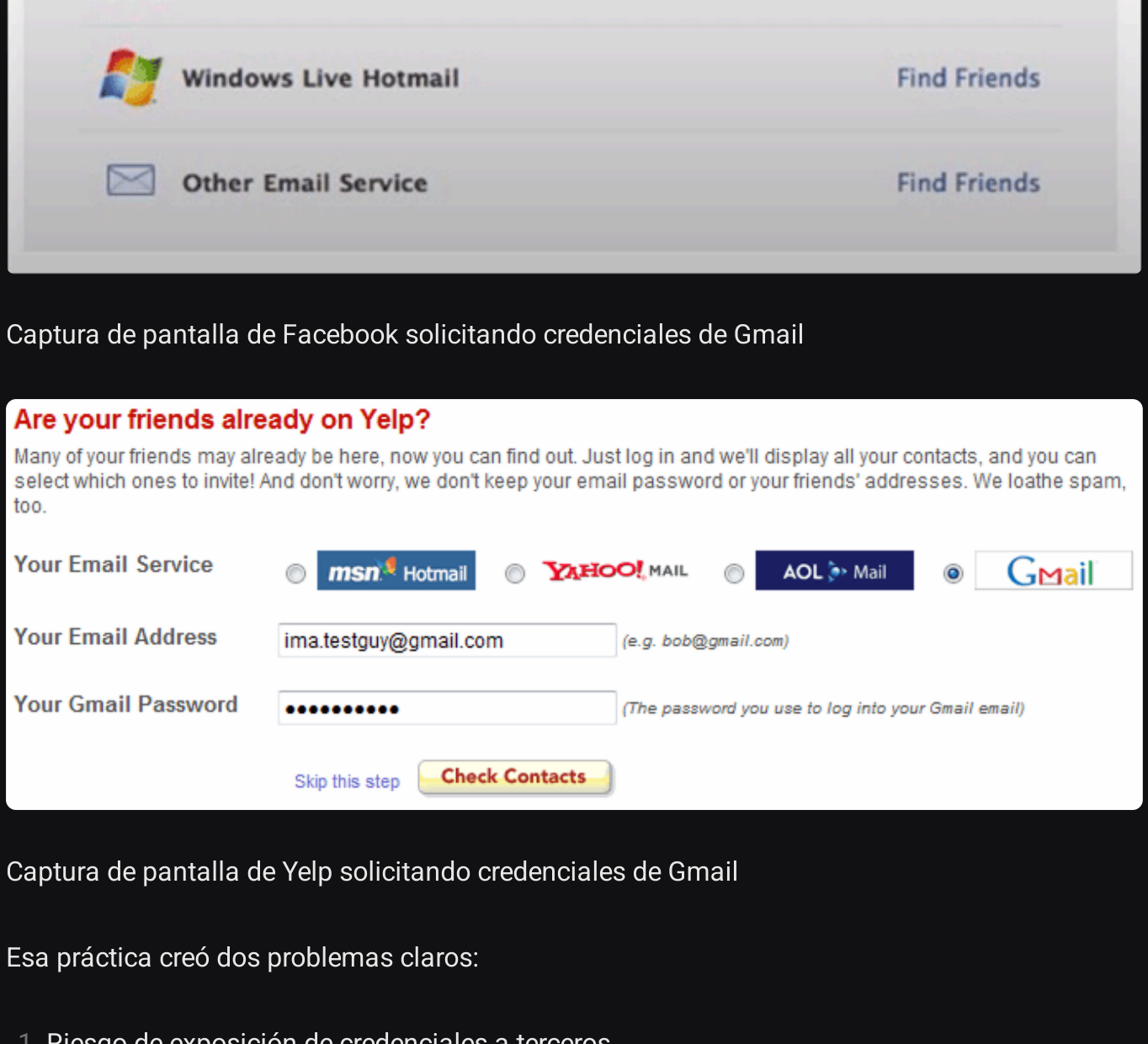
1 October 2025

Durante las últimas semanas he estado investigando a fondo cómo funcionan realmente **OAuth 2.0 y OIDC**. Son dos piezas clave de la seguridad en Internet, pero al mismo tiempo suelen estar mal explicadas y se confunden con facilidad. En este post quiero ordenar todas esas ideas y contarte, de forma clara y con ejemplos, cómo pasamos del modelo clásico de usuario/contraseña a los protocolos modernos de autorización y autenticación que usamos hoy en día.

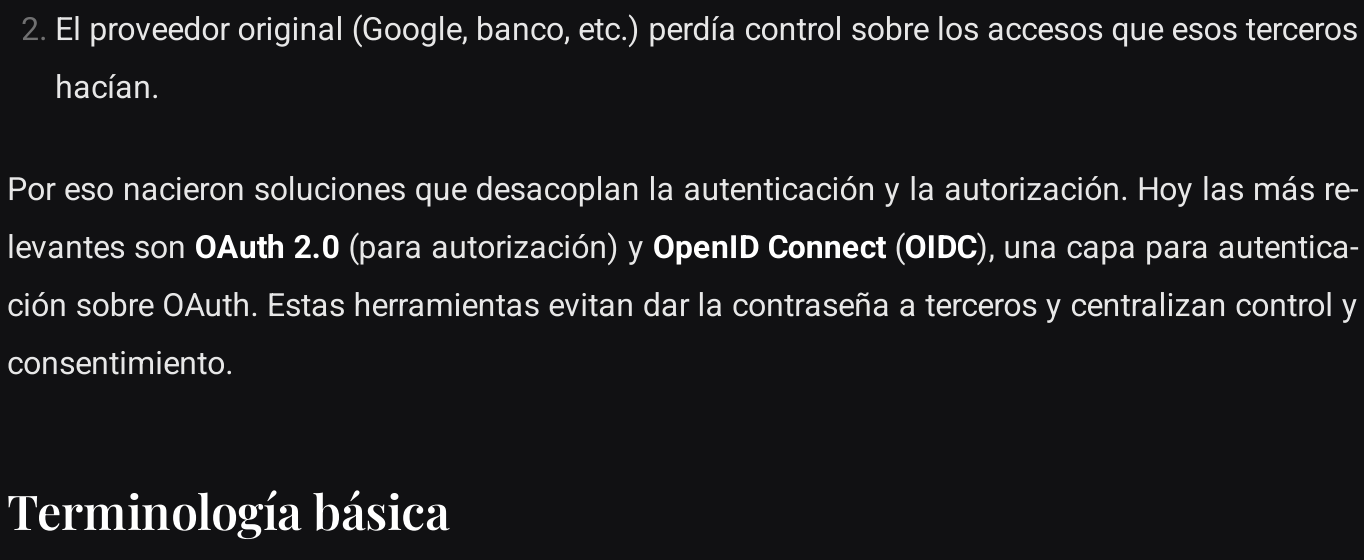
Del user/pass al nacimiento de OAuth

Al comienzo de Internet la única* forma de autorizar a un usuario en un servicio era pedirle su **usuario y contraseña**. Esto es tan antiguo como la web dinámica en PHP o ASP. El servidor guardaba un hash de la contraseña en la base de datos, si el hash coincidía, la aplicación creaba una sesión y mandaba una cookie con un session_id al navegador. Punto.

El problema era que tú, como desarrollador del servicio, eras el **responsable de todo el sistema de autenticación**. Tenías que asegurar el almacenamiento, la lógica de contraseñas, la regeneración de sesiones, las cookies, etc. Un fallo y se filtraban contraseñas. Peor aún: muchas apps de terceros pedían la contraseña del usuario para acceder a sus recursos en otros servicios. **Yelp** y **Facebook**, en sus inicios te pedían las credenciales de tu Gmail para "traer contactos". Las apps de gestión de finanzas almacenaban claves bancarias para leer movimientos. Era inseguro y era habitual.



Captura de pantalla de Facebook solicitando credenciales de Gmail



Captura de pantalla de Yelp solicitando credenciales de Gmail

Esa práctica creó dos problemas claros:

1. Riesgo de exposición de credenciales a terceros.
2. El proveedor original (Google, banco, etc.) perdía control sobre los accesos que esos terceros hacían.

Por eso nacieron soluciones que desacoplan la autenticación y la autorización. Hoy las más relevantes son **OAuth 2.0** (para autorización) y **OpenID Connect** (**OIDC**), una capa para autenticación sobre OAuth. Estas herramientas evitan dar la contraseña a terceros y centralizan control y consentimiento.

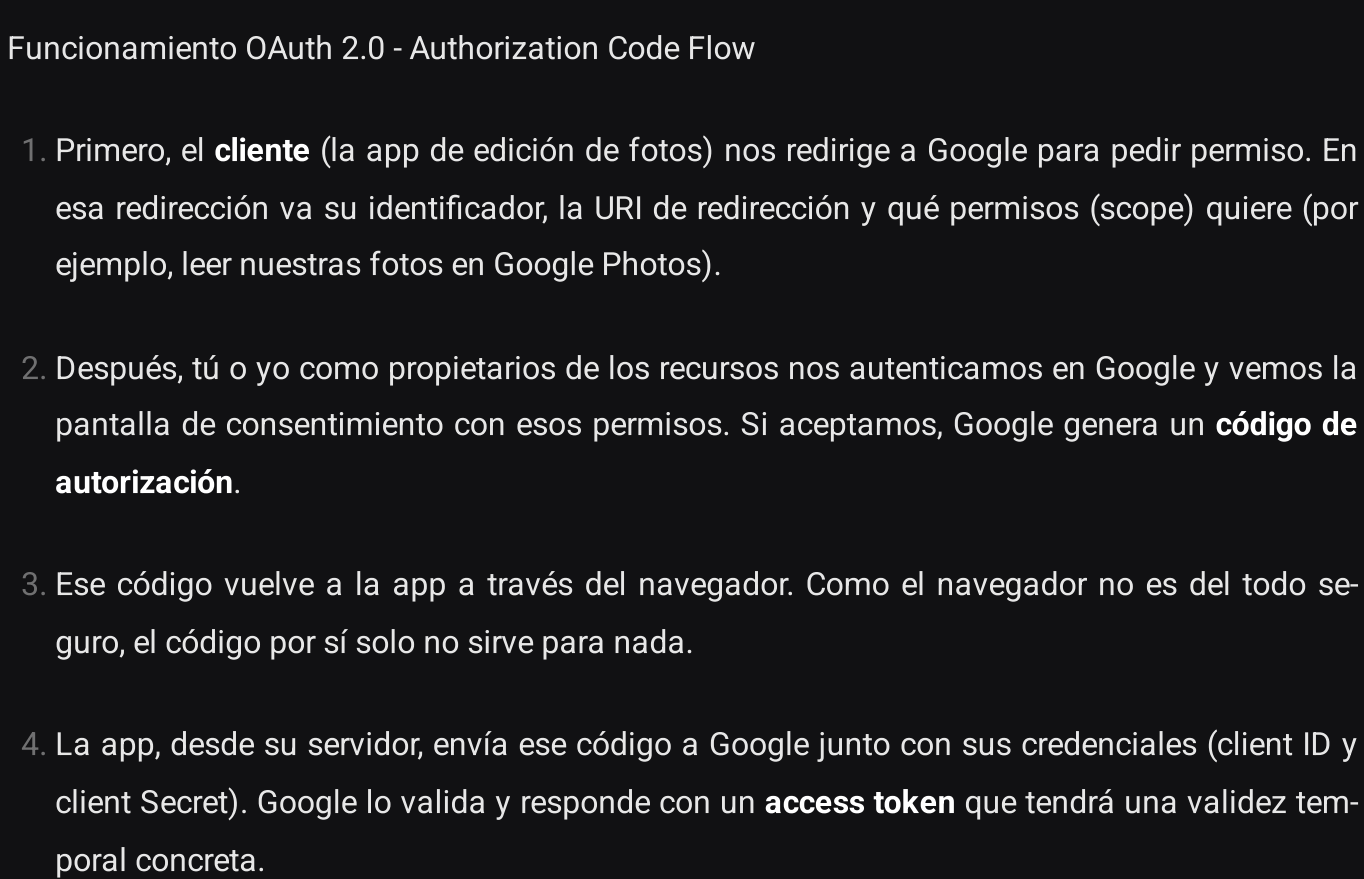
Terminología básica

A lo largo de todo el flujo de OAuth aparecen términos que pueden sonar abstractos. Antes de meternos en el detalle conviene aclararlos para no perdernos.

- **Resource Owner** (poseedor del recurso): el usuario que posee los datos (tus fotos en Google Photos por ejemplo).
- **Client** (cliente): la app que quiere acceder a los recursos (por ejemplo, un editor de fotos web).
- **Authorization Server** (servidor de autorización): el servicio que autentica al usuario y emite tokens (por ejemplo accounts.google.com).
- **Resource Server** (servidor de recursos): el servicio que guarda los recursos y acepta tokens para autorizarlos (photos.google.com). A menudo coincide con el authorization server pero no tiene que.
- **Authorization Grant**: la autorización que concede el usuario al cliente (el "consentimiento").
- **Redirect URI**: la URL a la que el authorization server redirige después de la autorización.
- **Access Token**: el token que el cliente usa para acceder al recurso. Suele ser un bearer token.
- **Scope**: lista de permisos solicitados (por ejemplo photos.read, photos.edit, email).

Funcionamiento de OAuth (Authorization Code Flow) paso a paso

En esta sección voy a tratar de la forma más sencilla posible el **Authorization Code Flow**, el flujo más común y seguro de **OAuth**. Lo explicaré paso a paso para que quede claro qué ocurre en cada fase, cómo viajan los datos entre navegador y servidor, y por qué este diseño separa lo visible al usuario del intercambio seguro en el back-channel. Para la explicación me apoyaré en un ejemplo en el que un usuario (tú o yo) queremos usar un servicio de edición de fotos online que quiere acceder a nuestras fotos de Google Photos.



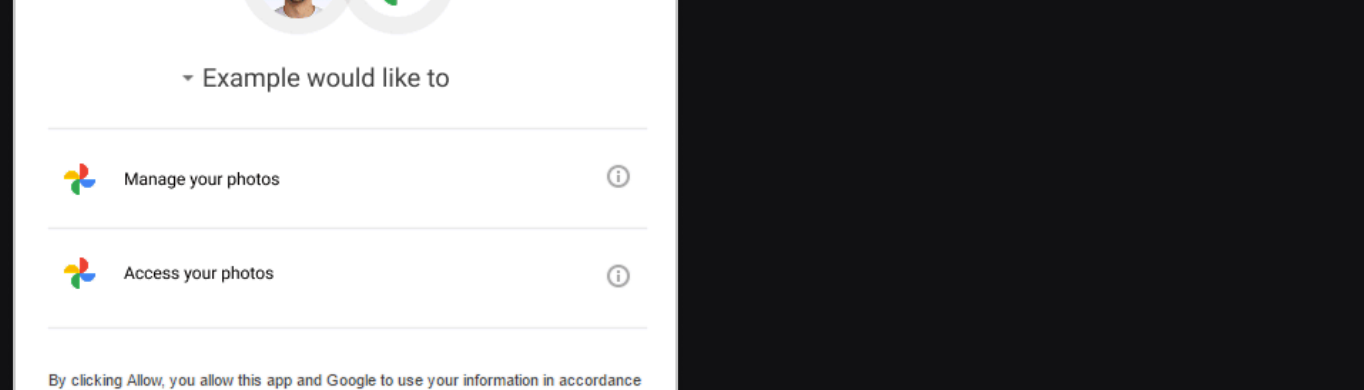
Funcionamiento OAuth 2.0 - Authorization Code Flow

1. Primero, el **cliente** (la app de edición de fotos) nos redirige a Google para pedir permiso. En esa redirección va su identificador, la URI de redirección y qué permisos (scope) quiere (por ejemplo, leer nuestras fotos en Google Photos).
2. Después, tú o yo como propietarios de los recursos nos autenticamos en Google y vemos la pantalla de consentimiento con esos permisos. Si aceptamos, Google genera un **código de autorización**.
3. Ese código vuelve a la app a través del navegador. Como el navegador no es del todo seguro, el código por sí solo no sirve para nada.
4. La app, desde su servidor, envía ese código a Google junto con sus credenciales (client ID y client Secret). Google lo valida y responde con un **access token** que tendrá una validez temporal concreta.
5. Con ese token, ahora sí, la app puede pedir a photos.google.com nuestras fotos y editarlas. Google valida el token y, si todo encaja, entrega los recursos.

Este diseño separa lo que pasa en el navegador (front-channel) de lo que se hace en servidor a servidor (back-channel). Las operaciones sensibles como la obtención del Token pasan por el back-channel. Más adelante te explico por qué el servidor de autenticación no nos devuelve directamente el Token.

Gestión de permisos: scopes y consentimiento

Los scopes permiten granularidad. Google puede exponer scopes para leer fotos, borrar fotos, leer contactos, enviar mail, etc. El cliente solicita solo lo que necesita. En la pantalla de consentimiento el usuario ve esos scopes y decide. **Principio simple: least privilege. Pide lo mínimo**. Evita scopes demasiado amplios como mail.modify o all.access. En el flujo que analizamos previamente deberíamos añadir el campo '**scope**' a la petición al servidor de autorización



Inclusión de Scopes en la petición al servidor de autorización

Históricamente Facebook fue permisivo y muchas webs obtenían permisos para publicar en muros o leer todo. Eso cambió por las filtraciones y por controles de consentimiento más estrictos. Hoy las plataformas exponen scopes bien definidos y revisan apps que piden demasiados permisos.

Ejemplo de scopes

¿Por qué un CODE y no el TOKEN desde el navegador?

Porque el navegador y la URL son front-channel. Son visibles para el usuario y para muchas superficies (history, logs de proxies, referrers). Si devuelves un access token directamente por la URL, puedes filtrar ese token. Un atacante podría hacer un man-in-the-middle o leer el token desde extensiones del navegador o capturas.

El **authorization code** es solo un identificador temporal que por sí mismo no da acceso a nada. La app lo canjea por un token en el **back-channel** (una petición segura servidor a servidor) usando sus credenciales: **client ID** y **client Secret**.

Estas credenciales no aparecen por arte de magia: cuando el desarrollador de la app de edición de fotos quiere integrarse con Google, primero debe **registrar la aplicación** en la consola de Google. En ese registro obtiene un client ID y un client Secret únicos.

El client Secret se guarda siempre en el backend de la aplicación, en un entorno seguro, nunca en el navegador ni en el código público. Cuando toca intercambiar el authorization code, la app hace un **HTTP POST** al servidor de Google, adjunta el código junto con su client ID y su client Secret, y Google responde con el **access token** válido para acceder a nuestras fotos.

Intercambio del código por el Token en OAuth 2.0

Tipos de Flujos en OAuth

En OAuth existen varios **flows** según el tipo de aplicación. El más común y el que hemos explicado es el **Authorization Code Flow**, que es el estándar y el más seguro.

También está el **Implicit Flow**, pensado para páginas que solo tienen front-end (por ejemplo, hechas en React o Next sin backend), aunque hoy en día no se recomienda porque expone directamente el token en el navegador.

Además, hay otros flows solo de backend que apenas se usan: el **Resource Owner Password Flow**, donde el usuario entrega usuario y contraseña (algo inseguro y obsoleto), y el **Client Credentials Flow**, utilizado únicamente en comunicaciones máquina a máquina.

OAuth no es autenticación. OIDC sí

OAuth 2.0 fue diseñado para **autorización**, no para autenticar. Es decir, OAuth contesta a "¿puede esta app acceder a este recurso?" y no a "¿quién es exactamente este usuario?".

OpenID Connect (OIDC) es una capa de **identidad** sobre OAuth 2.0. OIDC que añade:

- El scope openid.
- Un id_token, que suele ser un JWT.
- Un endpoint userinfo para pedir información del usuario con un access token.

El id_token es un JWT con 3 partes: header, payload (claims) y signature. En el payload encontrarás claims como sub (subject), iss (issuer), aud (audience), exp (expiry), iat (issued at), y claims de usuario name, email, etc. Puedes decodificar cualquier JWT en jwt.io para ver su contenido. OIDC también introduce el parámetro nonce para evitar replay attacks.

OIDC no sustituye a OAuth. Lo complementa. Si quieres que una app "loguee" al usuario con Google, lo correcto es usar OIDC. Muchas plataformas (Google, Facebook) exponen endpoints OIDC para ese fin. Si ves "Login with Google", bajo el capó suele usarse OIDC.

Entonces… **¿cuándo debo usar OAuth y cuándo OIDC?**

- Usa **OAuth** para tareas de **Autorización** como conceder acceso a tu API u obtener acceso a los datos de usuario en otros sistemas.
- Por otro lado, usa **OIDC** para tareas de **Autenticación** como inicios de sesión de usuarios o hacer que tus cuentas estén disponibles en otros sistemas.

OAuth2.0 resolvió el gran problema de la autorización: cómo dar acceso a aplicaciones sin entregarles nuestras contraseñas. **OpenID Connect** añadió la capa que faltaba, la identidad, permitiendo autenticar usuarios sobre el mismo protocolo. Esa diferencia es clave: **OAuth es autorización, OIDC es autenticación**.

Llevo un par de semanas profundizando en estos conceptos y ordenando todo lo que hay alrededor, porque las explicaciones en Internet suelen ser confusas y hasta contradictorias. Una de las mejores que he encontrado, y la que he seguido para estructurar este post, es la de Oktadev en este vídeo que te recomiendo ver.