Programación IV

Profesora: Cynthia Estrada
Fecha: 07/10
Tema: *Documentación API*

Integrantes:

- Díaz Rossini Juan José
- Gallo Genaro
- Navarro Victor Leandro

---

1) ¿Qué es OAuth y para qué se utiliza?

OAuth (Open Authorization) es un estándar abierto para **delegar autorización** sin compartir credenciales (usuario/contraseña). Permite que una aplicación (cliente) obtenga acceso limitado a recursos protegidos en nombre de un usuario, mediante tokens emitidos por un proveedor de autorización. Se usa para casos como permitir que una app acceda a tu calendario de Google o publicar en Twitter sin darle tu contraseña: el usuario autoriza y el servicio entrega tokens que la app usa para acceder.

---

2) ¿Cuáles son los principales actores involucrados en el proceso de OAuth?

1. **Resource Owner (Propietario del recurso):** el usuario que posee los datos.

2. **Client (Cliente):** la aplicación que solicita acceso (p. ej. una app móvil).

3. **Authorization Server (Servidor de autorización):** emite tokens después de autenticar y autorizar al usuario.

4. **Resource Server (Servidor de recursos):** aloja los datos protegidos y valida los tokens para permitir acceso.
   A menudo el Authorization Server y el Resource Server pueden ser la misma entidad, pero conceptualmente están separados.

---

3) ¿Qué es un token de acceso y cómo se utiliza en OAuth?

Un **token de acceso (access token)** es una credencial corta (habitualmente un JWT o un identificador opaco) que representa la autorización concedida al cliente para acceder a ciertos recursos durante un tiempo limitado. El cliente lo adjunta en cada petición al Resource Server (por ejemplo `Authorization: Bearer <access_token>`) y el servidor valida ese token para permitir o denegar la operación. No debe contener secretos del usuario y su alcance (scopes) y tiempo de vida son restricciones importantes.

---

4) ¿Qué es un token de actualización y cómo se utiliza en OAuth?

El **token de actualización (refresh token)** es una credencial de más larga duración que permite al cliente solicitar nuevos access tokens sin volver a pedir al usuario que inicie sesión. Se usa cuando el access token expira: el cliente hace una petición al Authorization Server con el refresh token para obtener un nuevo access token (y a veces un nuevo refresh token). Por seguridad, los refresh tokens normalmente se emiten sólo a clientes confidenciales (p. ej. servidores) y no a clientes públicos (p. ej. apps SPA sin backend).

---

5) ¿Cuál es la diferencia entre autenticación y autorización en OAuth?

- **Autenticación:** comprobar quién es el usuario (ej.: inicio de sesión con usuario/contraseña, 2FA).

- **Autorización:** decidir qué recursos o acciones puede realizar ese usuario o cliente (ej.: "permitir leer calendario" pero no borrar eventos).
  OAuth **no es un protocolo de autenticación** por sí mismo; es para autorización. Cuando se necesita autenticación estandarizada sobre OAuth, se usa **OpenID Connect** (OIDC), que añade identidad al flujo.

---

6) ¿Cuáles son los diferentes tipos de flujos de OAuth?

Los flujos más comunes (también llamados *grant types*):

- **Authorization Code Grant (Código de autorización):** para apps con backend seguro; más seguro porque el token se obtiene desde el servidor.

- **Implicit Grant (implícito):** antiguo, para SPAs; hoy en desuso por problemas de seguridad.

- **Client Credentials Grant (Credenciales de cliente):** para comunicación server-to-server (sin usuario).

- **Resource Owner Password Credentials (ROPC):** usuario pasa credenciales al cliente; no recomendado.

- **Refresh Token Grant:** para renovar access tokens.
  Además, OIDC añade flujos híbridos. En 2025 la recomendación general es usar **Authorization Code + PKCE** para clientes públicos y Authorization Code para clientes confidenciales.

---

7) ¿Cómo funciona el flujo de autorización de código de autorización en OAuth?

Resumen paso a paso (versión segura con PKCE para clientes públicos):

1. El cliente redirige al usuario al Authorization Server con `response_type=code`, `client_id`, `redirect_uri`, `scope` y un `state` (anti-CSRF). Con PKCE se añade `code_challenge`.

2. El usuario se autentica y acepta permisos.

3. El Authorization Server redirige al `redirect_uri` con un `code` (código de autorización) y el `state`.

4. El cliente (servidor/backend) intercambia ese `code` por un access token (y opcionalmente refresh token) en una petición directa al Authorization Server, autenticando el cliente (y en PKCE enviando `code_verifier`).

5. El Authorization Server devuelve tokens; el cliente usa el access token para llamar al Resource Server.
   Ventaja: el intercambio del token ocurre en el servidor, manteniendo secretos fuera del navegador.

---

8) ¿Cómo funciona el flujo de credenciales de cliente en OAuth?

En **Client Credentials Grant** el cliente se autentica directamente con el Authorization Server usando sus credenciales (client_id + client_secret) sin intervención de un usuario. Se utiliza para comunicaciones **server-to-server** donde la aplicación actúa en su propio nombre (p. ej. microservicio que necesita acceder a otro servicio). El servidor devuelve un access token que el cliente usa para llamar al Resource Server. No hay refresh token en muchos casos, y los scopes son del cliente, no del usuario.

---

9) ¿Cuál es el propósito del parámetro `redirect_uri` en OAuth?

`redirect_uri` es la URL a la que el Authorization Server enviará la respuesta (código de autorización o tokens). Sirve para:

- **Completar el flujo** devolviendo el código/token al cliente.

- **Seguridad:** el Authorization Server debe validar que el `redirect_uri` coincide con uno pre-registrado para ese `client_id`, evitando redirecciones a dominios maliciosos (open redirect attacks).
  Siempre usar URIs exactas o con reglas estrictas; para aplicaciones móviles se recomiendan URIs registradas (p. ej. esquemas personalizados o App Links) y para web URLs HTTPS.

---

10) ¿Qué es un token de acceso y cómo se protege?

(Ampliando lo dicho antes, con foco en protección)
Un **access token** autoriza acceso a recursos. Para protegerlo:

- **Transporte seguro:** siempre enviar por HTTPS (`TLS`) para evitar intercepción.
- **Almacenamiento seguro:** en servidores guardarlo en bases seguras; en clientes web evitar localStorage si posible (usar cookies `HttpOnly` y `Secure` con SameSite apropiado). En móviles usar almacenamiento seguro del sistema (Keychain/Keystore).
- **Vida corta:** emitir access tokens de corta duración para limitar impacto si se filtran.
- **Scopes y principals mínimos:** limitar privilegios al mínimo necesario (principio de least privilege).
- **Rotación y revocación:** poder revocar tokens y usar refresh tokens con caducidad/rotación.
- **Validación en Resource Server:** validar firma, issuer, audience, scopes y tiempo de expiración.
- **Protección contra CSRF y XSS:** usar `state`, PKCE y evitar exponer tokens a JavaScript cuando sea posible.

---

11) ¿Qué es un token de actualización y cómo se protege?

Un **token de actualización** (Refresh Token) es una credencial que permite obtener **nuevos tokens de acceso** sin que el usuario tenga que autenticarse nuevamente.
Generalmente tiene una **vida útil más larga** que el token de acceso, pero también implica **más riesgo** si se filtra.
Para protegerlo:

- Se debe **almacenar sólo en el servidor** o en un entorno seguro (por ejemplo, almacenamiento cifrado en móvil).
- **Nunca debe enviarse al frontend en texto plano**.
- Usar **HTTPS** para todas las solicitudes.
- Asociarlo a un **cliente, usuario y dispositivo específicos**, para evitar su reutilización en otros contextos.

---

12) ¿Cómo se manejan los tokens de acceso caducados en Oauth?

Cuando un **token de acceso caduca**, la aplicación cliente:

1. Detecta que el token ya no es válido (por ejemplo, recibe un error 401 “Unauthorized”).
2. Envía una solicitud al **servidor de autorización** con el **refresh token** para obtener uno nuevo.
3. Si el refresh token también expiró o fue revocado, el usuario deberá **autenticarse nuevamente**.
   Esto permite mantener la sesión activa sin pedirle al usuario su contraseña todo el tiempo.

---

13) ¿Qué es la revocación de tokens y cómo se implementa en Oauth?

La **revocación de tokens** es el proceso de **invalidar manualmente** un token antes de su expiración, por ejemplo si el usuario cierra sesión o se detecta actividad sospechosa.
Se implementa mediante un **endpoint de revocación** definido en el estándar OAuth 2.0 (`/revoke`), al cual el cliente envía el token a invalidar.
El servidor lo marca como **revocado en la base de datos** o lista negra, impidiendo su uso futuro.

---

14) ¿Cuáles son las mejores prácticas para proteger los tokens de acceso y actualización?

- **Usar HTTPS** siempre.
- **Evitar almacenar tokens en `localStorage` o `sessionStorage`**, preferir cookies seguras con `HttpOnly` y `SameSite`.
- **Cifrar los tokens** si se guardan en base de datos.
- **Rotar los refresh tokens** (emitir uno nuevo cada vez que se usa).
- **Asignar tiempos de expiración cortos** a los tokens de acceso.
- **Limitar el alcance (scope)** del token al mínimo necesario.
- **Implementar revocación** efectiva y auditorías de acceso.

---

15) ¿Cómo se implementa Oauth en una aplicación web?

En una aplicación web, OAuth se implementa generalmente con el **flujo de código de autorización**:

1. El usuario inicia sesión en el **proveedor de identidad** (Google, GitHub, etc.).
2. El proveedor redirige a la app con un **código de autorización**.
3. La app intercambia ese código por un **token de acceso** y opcionalmente un **refresh token**.
4. El backend usa el token para acceder a la API en nombre del usuario.
   Todo esto se realiza del lado del **servidor**, para mantener seguros los tokens.

---

16) ¿Cómo se utiliza Oauth en una aplicación móvil?

En móviles, se utiliza un **flujo especial adaptado**, generalmente:

- **Authorization Code con PKCE (Proof Key for Code Exchange)**.
  Este método añade seguridad evitando ataques de interceptación, ya que no se usa un “client secret” (difícil de proteger en apps móviles).
  El usuario se autentica mediante el navegador del dispositivo, se obtiene un código temporal, y la app lo intercambia por un token usando PKCE.

---

17) ¿Cuáles son las ventajas y desventajas de utilizar Oauth en una aplicación?

Ventajas:

- Evita que las apps manejen directamente las contraseñas de los usuarios.
- Permite acceso controlado a recursos de terceros (por ejemplo, usar tu cuenta de Google).
- Mejora la seguridad y experiencia de usuario.
- Facilita el inicio de sesión único (SSO).

Desventajas:

- Implementación compleja y requiere configuración cuidadosa.
- Si se usa mal, puede generar **vulnerabilidades graves** (por ejemplo, fugas de tokens).
- Depende de servicios externos (proveedores de identidad).

---

18) ¿Cómo se manejan los errores y excepciones en Oauth?

Los errores se manejan con **respuestas estandarizadas**, incluyendo un código de error y una descripción.
Ejemplos:

- `invalid_request`: parámetro incorrecto o ausente.
- `invalid_client`: credenciales del cliente inválidas.
- `invalid_grant`: el código de autorización es inválido o expiró.
- `unauthorized_client`: el cliente no tiene permiso para usar ese flujo.
  El cliente debe **interpretar el error y responder apropiadamente**, por ejemplo, redirigiendo al usuario a iniciar sesión de nuevo o mostrando un mensaje de error claro.

---

19) ¿Cuáles son las mejores prácticas para documentar y comunicar la implementación de Oauth en una aplicación?

- Describir **claramente los flujos utilizados** (Authorization Code, Client Credentials, etc.).
- Incluir **diagramas de secuencia** de los pasos y endpoints.
- Documentar los **scopes disponibles** y su propósito.
- Explicar **cómo renovar y revocar tokens**.
- Detallar la **seguridad aplicada** (PKCE, HTTPS, rotación de tokens).
- Mantener documentación actualizada para desarrolladores y auditores de seguridad.

---

20) ¿Cómo se utiliza Oauth en un escenario de inicio de sesión único (SSO)?

En un entorno de **Single Sign-On (SSO)**, OAuth permite que el usuario **se autentique una vez** en un proveedor (por ejemplo, Google o la empresa), y acceda a **múltiples aplicaciones conectadas** sin volver a iniciar sesión.
El **proveedor de identidad** maneja la autenticación centralizada y emite tokens de acceso válidos para cada aplicación.
De esta forma, se simplifica la gestión de credenciales y se mejora la seguridad, ya que **todas las apps confían en el mismo servidor de autorización.**

---

21) ¿Cómo se utiliza Oauth en un escenario de API de terceros?

En un escenario de **API de terceros**, OAuth se utiliza para permitir que una aplicación acceda a los recursos de un usuario **sin compartir sus credenciales**.
Ejemplo: una app de calendario que accede a tu cuenta de Google Calendar.
El proceso es:

1. El usuario autoriza a la app a acceder a su cuenta.
2. La app obtiene un **token de acceso** emitido por el servidor de Google.
3. Ese token se usa en cada solicitud a la API.
   De esta forma, la app puede actuar en nombre del usuario sin tener su contraseña, y el usuario puede revocar el acceso cuando quiera.

---

22) ¿Cómo se utiliza Oauth en un escenario de microservicios?

En una arquitectura de **microservicios**, OAuth permite centralizar la autenticación y autorización:

- Un **servidor de autorización** emite tokens JWT (JSON Web Tokens).

- Cada microservicio valida el token sin necesidad de comunicarse directamente con el servidor de autenticación.
  Esto mejora la **escalabilidad y seguridad**, porque cada microservicio sólo necesita verificar la firma y los permisos del token.
  Además, se pueden definir **scopes o roles específicos** para limitar el acceso entre servicios.

---

23) ¿Cómo se maneja la autorización y autenticación en un escenario de Oauth con múltiples clientes?

Cuando hay **múltiples clientes** (por ejemplo, web, móvil, desktop), cada uno debe tener su propia **identidad de cliente** registrada en el servidor de autorización.
Cada cliente:

- Obtiene su propio **Client ID** (y a veces **Client Secret**, si es confidencial).

- Usa el flujo OAuth adecuado (web: código de autorización, móvil: PKCE).
  El servidor de autorización mantiene control de **qué cliente está usando qué token**, lo que permite revocar permisos por cliente sin afectar a los demás.
  Esto también evita que una app móvil, por ejemplo, use tokens emitidos para la versión web.

---

24) ¿Cuáles son los desafíos y consideraciones al implementar Oauth en un entorno de producción?

Los principales desafíos incluyen:

- **Seguridad de los tokens** (evitar su robo o reutilización).

- **Protección contra ataques CSRF, XSS o de redirección abierta.**

- **Configuración correcta del redirect_uri** y scopes.

- **Gestión de expiraciones y revocaciones**.

- **Escalabilidad del servidor de autorización.**

- **Cumplimiento normativo** (por ejemplo, RGPD si se procesan datos personales).
  Una mala configuración de OAuth puede abrir vulnerabilidades graves, por lo que se deben aplicar auditorías y pruebas de penetración antes del despliegue.

---

25) ¿Cómo se implementa Oauth con OpenID Connect?

OpenID Connect (OIDC) es una capa que se construye sobre OAuth 2.0 para agregar **autenticación de usuario**, no solo autorización.
Incluye un nuevo tipo de token: el **ID Token**, que contiene información sobre el usuario autenticado (nombre, email, etc.) en formato JWT.
Implementación típica:

1. El cliente solicita autorización.

2. El servidor OIDC devuelve un **ID Token** y un **Access Token**.

3. El ID Token identifica al usuario, mientras que el Access Token se usa para acceder a APIs.
   Con OIDC, las apps pueden autenticar usuarios y obtener datos básicos de su perfil sin manejar contraseñas directamente.

---

26) ¿Cómo se maneja la autenticación y autorización en un escenario de Oauth con múltiples proveedores de identidad?

Cuando hay **múltiples proveedores de identidad** (Google, Microsoft, GitHub, etc.), la aplicación actúa como un **cliente de varios servidores de autorización**.
Cada proveedor tiene su propio flujo y endpoint de tokens, por lo que la app:

- Redirige al usuario al proveedor elegido.

- Recibe el token de ese proveedor.

- Normaliza la información recibida (por ejemplo, adaptando el formato de perfil).
  Para simplificar la integración, muchas apps usan bibliotecas o gateways como **Auth0, Keycloak o Firebase Authentication**, que gestionan múltiples proveedores desde una interfaz unificada.

---

27) ¿Cuáles son las implicaciones de seguridad de utilizar Oauth en un entorno de nube?

En la nube, el uso de OAuth requiere medidas adicionales:

- **Aislar los secretos del cliente** en servicios seguros (por ejemplo, AWS Secrets Manager o GCP Secret Manager).

- Usar **roles y permisos granulares** en lugar de tokens con privilegios amplios.

- **Cifrar tokens en tránsito y en reposo.**

- **Monitorear el uso de tokens** para detectar accesos sospechosos.

- Implementar **rotación de claves y certificados** periódicamente.
  Las credenciales expuestas en entornos compartidos o repositorios pueden comprometer toda la

infraestructura si no se gestionan correctamente.

---

28) ¿Cómo se maneja la revocación de tokens en un escenario de Oauth con múltiples clientes?

Cuando hay varios clientes conectados, la revocación debe gestionarse de forma **centralizada**:

- El servidor de autorización mantiene una lista de **tokens activos** asociados a cada cliente.
- Cuando el usuario revoca el acceso, se invalidan **todos los tokens relacionados** o solo los de un cliente específico.
- Cada microservicio o API debe **verificar el estado del token** periódicamente o validar su firma con una lista de revocación.
  Esto evita que un token revocado siga siendo aceptado en otras partes del sistema.

---

29) ¿Cuáles son las mejores prácticas para realizar auditorías y pruebas de seguridad en una implementación de Oauth?

- **Registrar todos los eventos de acceso y autorización** (emisión, renovación, revocación).
- **Analizar los logs** para detectar patrones de abuso o tokens anómalos.
- **Probar escenarios de error y redirecciones maliciosas.**
- **Validar la seguridad de los endpoints** (token, revoke, introspect).
- **Hacer pruebas de penetración (pentesting)** centradas en OAuth y JWT.
- **Revisar los tiempos de expiración y scopes.**
- **Verificar el cumplimiento de las RFCs de OAuth 2.0 y OpenID Connect.**
  Una auditoría continua asegura que la implementación siga siendo segura frente a ataques y errores de configuración.