OAuth 2.0 - Complete Technical Notes

---

1. Introduction

Definition

- **OAuth 2.0** is an **authorization framework** (not authentication)
- Enables **third-party applications** to obtain **limited access** to user resources
- Without sharing user credentials (username/password)
- Industry standard protocol defined in **RFC 6749** (October 2012)

Origin

- Developed by **IETF OAuth Working Group**
- Released as OAuth 1.0 in 2010, OAuth 2.0 in 2012
- Created to replace proprietary authorization methods
- Major contributors: Google, Microsoft, Facebook, Twitter

---

2. Why We Need OAuth 2.0?

Problem Statement

- **Traditional Approach**: Apps needed your username/password to access your data
- **Issues**:
  - Security risk: Sharing credentials with third parties
  - No granular control: All-or-nothing access
  - Difficult to revoke: Had to change password everywhere
  - Trust issues: Every app has full account access

Real-World Analogy: The Valet Key Problem

Scenario: You arrive at a luxury hotel with your car

Old Method (Without OAuth):

- You give the **master key** to the valet

- Valet can: open trunk, glove box, drive anywhere, access everything
- **Risk**: Complete access to your vehicle

New Method (With OAuth):

- You give a **valet key** (limited access token)
- Valet can: only park the car, limited distance
- Cannot: open trunk, glove box, or drive beyond parking area
- **Benefit**: Controlled, limited, revocable access

What OAuth 2.0 Solves

- **Delegated Access**: Apps access resources on your behalf
- **No Password Sharing**: Apps never see your credentials
- **Limited Scope**: Apps get only requested permissions
- **Revocable**: You can revoke access anytime
- **Secure**: Token-based system with expiration

---

3. Comparison: OAuth 1.0 vs OAuth 2.0

Feature	OAuth 1.0	OAuth 2.0
Complexity	Complex cryptographic signatures required	Simplified, uses bearer tokens
HTTPS Requirement	Optional (uses signatures)	**Mandatory** for security
Token Types	Single token type	Multiple: Access, Refresh tokens
Client Types	Limited support	Web, mobile, native, SPA support
Token Expiration	Long-lived tokens	Short-lived access + refresh tokens
Flow Types	3 flows	**6 grant types** (more flexible)
Mobile Support	Poor	**Excellent** (designed for it)
API Calls	Requires signing each request	Simple bearer token in header
Performance	Slower (crypto overhead)	**Faster** (no signatures)
User Experience	Multiple redirects	Streamlined UX

Feature	OAuth 1.0	OAuth 2.0
Security	Built-in signature security	Relies on **TLS/HTTPS**
Adoption	Declining	**Industry standard**

4. Key Terminology & Roles

Core Roles

1. Resource Owner (User)

- The **person** who owns the data
- Example: You (the Facebook user)
- Has authority to grant access to their resources

2. Client (Application)

- The **third-party application** requesting access
- Example: "Photo Printing App" wanting your Facebook photos
- Types: Confidential (server-side) or Public (mobile/SPA)

3. Authorization Server

- **Issues tokens** after successful authentication
- Validates user credentials
- Example: Facebook's OAuth server at `oauth.facebook.com`
- Handles authorization requests and consent

4. Resource Server

- Hosts the **protected resources** (data/APIs)
- Validates access tokens
- Example: Facebook Graph API serving your photos
- Can be same as or separate from Authorization Server

Key Terms

Access Token

- Short-lived credential (typically **15-60 minutes**)
- Used to access protected resources
- Format: Random string or **JWT** (JSON Web Token)
- Sent with each API request

Refresh Token

- Long-lived credential (days/months)
- Used to obtain **new access tokens**
- More secure, stored safely
- Not sent to Resource Server

Scope

- Defines **permissions** requested by client
- Example: `read:profile`, `write:posts`, `read:photos`
- User sees these during consent screen
- Granular access control

Authorization Code

- Temporary code exchanged for tokens
- **Single-use**, short expiration (~10 minutes)
- Used in Authorization Code flow

Redirect URI

- URL where user is sent after authorization
- Must be **pre-registered** with Authorization Server
- Security measure against token theft

Client ID & Client Secret

- **Client ID**: Public identifier for the application
- **Client Secret**: Confidential password (for server apps only)
- Used to authenticate the client application

---

5. Architecture & Flow

High-Level OAuth 2.0 Flow

```
Step 1: Authorization Request
User → Client App → Authorization Server
"I want to use Photo App with my Facebook photos"

Step 2: User Authentication & Consent
Authorization Server → User
"Do you allow Photo App to access your photos?"

Step 3: Authorization Grant
User (Approves) → Authorization Server → Client App
"Here's an authorization code"

Step 4: Token Request
Client App → Authorization Server (with code + client credentials)
"Exchange code for access token"

Step 5: Access Token Response
Authorization Server → Client App
"Here's your access token & refresh token"

Step 6: Access Protected Resource
Client App → Resource Server (with access token)
"Give me user's photos using this token"

Step 7: Protected Resource Response
Resource Server (validates token) → Client App
"Here are the photos (JSON/data)"
```

Detailed Step-by-Step Process

Step 1: Authorization Request

- Client redirects user to Authorization Server
- Includes: `client_id`, `redirect_uri`, `scope`, `state`
- Example: `https://auth.example.com/authorize?client_id=ABC&redirect_uri=...&scope=read:profile`

Step 2: User Login & Consent

- User logs into Authorization Server (if not already)
- Sees consent screen showing requested permissions

- User approves or denies access

Step 3: Authorization Code Issued

- Authorization Server redirects to `redirect_uri`
- Includes authorization code in URL
- Example: `https://client.app/callback?code=XYZ123&state=...`

Step 4: Exchange Code for Token

- Client sends **POST request** to token endpoint
- Includes: `code`, `client_id`, `client_secret`, `redirect_uri`
- This happens **server-to-server** (not in browser)

Step 5: Receive Tokens

- Authorization Server validates code and credentials
- Responds with JSON:

```json
{
  "access_token": "eyJhbGc...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOk...",
  "scope": "read:profile"
}
```

Step 6: Access Protected API

- Client sends request to Resource Server
- Includes access token in header:

```
Authorization: Bearer eyJhbGc...
```

Step 7: Token Validation & Response

- Resource Server validates token (signature, expiration, scope)
- Returns requested data if valid
- Returns `401 Unauthorized` if invalid/expired

---

6. Grant Types (Authorization Flows)

6.1 Authorization Code Grant

Description:

- Most **secure** and commonly used flow
- Involves browser redirect + backend token exchange
- Uses **authorization code** as intermediate step

Flow:

1. Client redirects user to Authorization Server
2. User authenticates and approves
3. Authorization code sent to client
4. Client exchanges code for tokens (server-side)

When to Use:

- ✅ **Web applications** with backend server
- ✅ When client secret can be kept confidential
- ✅ Apps that need refresh tokens
- ✅ **Best security** practice for user-facing apps

Example: Traditional web apps like Dropbox, Gmail clients

6.2 Authorization Code with PKCE

Description:

- Extension of Authorization Code flow
- **PKCE** = Proof Key for Code Exchange (RFC 7636)
- Adds extra security layer for public clients

Additional Steps:

1. Client generates random **code_verifier** (43-128 chars)
2. Creates **code_challenge** = SHA256(code_verifier)

3. Sends `code_challenge` with authorization request
4. Sends `code_verifier` with token request
5. Server validates: SHA256(code_verifier) == code_challenge

When to Use:

- ✅ **Mobile applications** (iOS, Android)
- ✅ **Single Page Applications** (React, Angular, Vue)
- ✅ **Desktop applications**
- ✅ Any app that **cannot keep client secret** secure
- ✅ **Recommended for ALL public clients**

Example: Mobile banking apps, Twitter mobile app

6.3 Implicit Grant (Deprecated)

Description:

- Simplified flow for browser-based apps
- Access token returned directly in URL fragment
- **No authorization code** step

Flow:

1. Client redirects to Authorization Server
2. User authenticates
3. Access token returned in URL: `#access_token=...`

When to Use:

- ❌ **Deprecated** - Do NOT use for new applications
- ❌ Replaced by **Authorization Code + PKCE**
- Security issues: token exposed in browser history

Why Deprecated:

- Tokens visible in browser history
- No refresh token support
- Vulnerable to token theft

6.4 Client Credentials Grant

Description:

- **Machine-to-machine** authentication
- No user involvement
- Client authenticates with own credentials

Flow:

1. Client sends `client_id` + `client_secret` to token endpoint
2. Receives access token directly
3. Uses token to access APIs

When to Use:

- ✅ **Server-to-server** communication
- ✅ **Backend services** accessing APIs
- ✅ **Microservices** authentication
- ✅ **Cron jobs** or scheduled tasks
- ✅ No user context needed

Example:

- Payment processor calling bank API
- Analytics service accessing data warehouse
- CI/CD pipeline accessing deployment APIs

6.5 Resource Owner Password Credentials (Legacy)

Description:

- User provides **username & password** directly to client
- Client exchanges credentials for tokens
- **Highly discouraged** in OAuth 2.0

Flow:

1. User enters credentials in client app
2. Client sends credentials to token endpoint
3. Receives access token

When to Use:

- ⚠️ **Only for trusted first-party apps**
- ⚠️ When redirect-based flow impossible
- ⚠️ Legacy system migration
- ❌ **Avoid whenever possible**

Why Avoid:

- Client sees user's password
- Defeats purpose of OAuth
- No two-factor authentication support

6.6 Device Authorization Grant (Device Flow)

Description:

- For devices with **limited input capabilities**
- User authorizes on separate device (phone/computer)

Flow:

1. Device requests device code from Authorization Server
2. Device displays code & URL to user
3. User visits URL on phone/computer and enters code
4. User authenticates and approves
5. Device polls token endpoint
6. Receives access token when approved

When to Use:

- ✅ **Smart TVs** (Netflix, YouTube login)
- ✅ **Gaming consoles** (Xbox, PlayStation)

- ✅ **IoT devices** (printers, cameras)
- ✅ **Command-line tools** (AWS CLI, Google Cloud SDK)

Example:

- "Visit youtube.com/activate and enter code: ABCD-EFGH"
- Smart home devices pairing

---

7. Security Best Practices

Token Types & Security

Access Token:

- **Lifetime**: Short (15-60 minutes)
- **Storage**: Memory (web) or secure storage (mobile)
- **Transmission**: Always over **HTTPS**
- **Format**: Opaque string or **JWT**

Refresh Token:

- **Lifetime**: Long (days to months)
- **Storage**: Secure, encrypted storage only
- **Rotation**: Should be rotated on each use
- **Revocation**: Can be revoked by user or admin

Security Best Practices

1. Always Use HTTPS

- **Mandatory** for all OAuth communications
- Prevents token interception
- Required by OAuth 2.0 spec

2. Validate Redirect URIs

- **Pre-register** all redirect URIs
- Exact match validation (no wildcards)
- Prevents authorization code theft

3. Use State Parameter

- Random value sent with auth request
- Validated on callback
- **Prevents CSRF attacks**

4. Implement PKCE

- Use for **all public clients** (mobile, SPA)
- Protects against code interception
- Now recommended for confidential clients too

5. Token Storage

- Web: Store in memory, not localStorage
- Mobile: Use **Keychain (iOS)** or **Keystore (Android)**
- Never expose tokens in logs or URLs

6. Scope Limitation

- Request **minimum necessary scopes**
- Implement principle of least privilege
- Allow users to review and limit scopes

7. Token Expiration

- Use short-lived access tokens
- Implement refresh token rotation
- Revoke tokens on logout

8. Client Secret Protection

- **Never** expose in frontend code
- Use environment variables
- Rotate secrets periodically

9. Input Validation

- Validate all authorization responses
- Check `state` parameter

- Verify token signatures (JWT)

10. Token Revocation

- Implement token revocation endpoint
- Allow users to revoke access
- Monitor for suspicious activity

---

8. Revision Table (Quick Reference)

Concept	Key Points
What is OAuth 2.0?	Authorization framework, NOT authentication. Allows third-party access without password sharing.
Main Problem Solved	Eliminates password sharing with third parties, enables delegated access.
4 Core Roles	Resource Owner (User), Client (App), Authorization Server, Resource Server
Token Types	Access Token (short-lived, 15-60 min), Refresh Token (long-lived, rotated)
Most Secure Flow	**Authorization Code + PKCE** (for all clients now)
Machine-to-Machine	**Client Credentials** grant (no user involved)
Limited Input Devices	**Device Flow** (Smart TV, IoT)
Deprecated Flows	Implicit Grant (use Auth Code + PKCE instead), Password Grant (avoid)
Critical Security	Always HTTPS, validate redirect_uri, use state parameter, implement PKCE
Scope	Defines permissions (read:profile, write:posts), shown to user in consent screen
PKCE	Proof Key for Code Exchange - adds security for public clients (mobile, SPA)
vs OAuth 1.0	Simpler (no signatures), faster, better mobile support, requires HTTPS
Token Storage	Memory (web), Keychain/Keystore (mobile), never localStorage
Common Use Cases	Social login, API access delegation, mobile apps, microservices

---

Flowchart for Drawing

Authorization Code Flow (Draw This)

```
┌─────────────┐
│   User      │
└──────┬──────┘
       │ 1. Clicks "Login with Provider"
       ▼
┌─────────────┐
│ Client App  │
└──────┬──────┘
       │ 2. Redirect to Auth Server
       │    (with client_id, scope, redirect_uri, state)
       ▼
┌─────────────────────┐
│ Authorization Server │
└──────┬──────────────┘
       │ 3. User Login & Consent Screen
       ▼
┌─────────────┐
│   User      │ (Approves)
└──────┬──────┘
       │ 4. Redirect to Client
       │    (with authorization code)
       ▼
┌─────────────┐
│ Client App  │
└──────┬──────┘
       │ 5. POST to Token Endpoint
       │    (code + client_secret)
       ▼
┌─────────────────────┐
│ Authorization Server │
└──────┬──────────────┘
       │ 6. Returns Access & Refresh Token
       ▼
┌─────────────┐
│ Client App  │
└──────┬──────┘
       │ 7. API Call with Access Token
       ▼
┌─────────────────┐
│ Resource Server │
└──────┬──────────┘
       │ 8. Returns Protected Data
       ▼
┌─────────────┐
```

```
│ Client App    │
└───────────────┘
```

Summary

OAuth 2.0 is the **industry standard** for secure, delegated authorization. It solves the critical problem of third-party access without credential sharing through a **token-based system**. The framework supports multiple grant types for different scenarios, with **Authorization Code + PKCE** being the recommended flow for most applications. Security depends on proper implementation: **HTTPS everywhere**, token management, and following best practices. Understanding the roles, flows, and security considerations is essential for building secure modern applications.

Remember: OAuth 2.0 is for **AUTHORIZATION** (what you can do), not **AUTHENTICATION** (who you are). For authentication, use **OpenID Connect** (built on top of OAuth 2.0).