OAuth 2.1 and OpenID Connect

Version OAuth 2.1 draft-17 · verified 2026-11-02

The OAuth flows a web developer actually builds — code + PKCE, client credentials, device — plus OIDC ID tokens, validation and BFF.

Contents

Mental model 3

Roles and the 2.1 baseline 3

Authorization code with PKCE 5

Token exchange and refresh 7

Client credentials and device flow 9

ID token vs access token 10

Discovery, JWKS and validation 12

Browser apps: the BFF pattern 13

Common misconfigurations 15

Further reading 16

Mental model

OAuth delegates *access*: a user lets a client call an API on their behalf without handing over their password. OpenID Connect adds *identity* on top: an ID token that tells the client who signed in. A token's audience and scopes say what it is for, not who holds it — whoever presents a bearer token gets its powers, so every check is about where a token came from and where it is allowed to go.

Roles and the 2.1 baseline

Role	Is
Resource owner	The user who grants access
Client	Your app asking for access
Authorization server (AS)	Issues tokens; the identity provider
Resource server (RS)	The API that accepts access tokens

OAuth 2.1 is still an IETF Internet-Draft, not an RFC: this sheet follows draft-17 (30 Oct 2026). It folds RFC 6749, PKCE and the RFC 9700 security advice into one document. What changes from 2.0:

- Authorization code requires PKCE, and the `plain` method is gone
- Implicit and password grants are removed
- Redirect URIs match by exact string
- No bearer tokens in query strings
- Public clients get sender-constrained or one-time-use refresh tokens

```
GET /orders/42 HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
```

> **Gotcha:** `response_type=token` (implicit) still works on many identity providers. It returns the access token in the URL fragment, into browser history and to every script on the page. Do not use it, even where it is enabled.

Authorization code with PKCE

Step	Who	Carries
1. Redirect to `/authorize`	Client	`code_challenge`, `state`
2. User signs in, consents	AS	—
3. Redirect back with `code`	AS	`code`, `state`, `iss`
4. POST to `/token`	Client	`code`, `code_verifier`

This is the only flow for a user signing in through a browser — web apps, SPAs and native apps alike. PKCE binds the code to the client that started the flow: the

client keeps a random verifier and sends only its SHA-256 hash, so a stolen code is useless without the verifier.

```
import { createHash, randomBytes as rb }
  from "node:crypto";
const verifier = rb(32).toString("base64url");
const challenge = createHash("sha256")
  .update(verifier).digest("base64url");
const q = new URLSearchParams({
  response_type: "code", client_id: "web-app",
  redirect_uri: "https://app.example.com/cb",
  scope: "openid profile orders:read",
  state: rb(16).toString("base64url"),
  code_challenge: challenge,
  code_challenge_method: "S256",
});
const url = `https://as.example.com/authorize?${q}
`;
```

> **Gotcha:** Check `state` on the callback and the `iss` parameter against the issuer you started with. Skipping `iss` leaves you open to mix-up attacks when you support more than one identity provider.

Token exchange and refresh

Parameter	Value
`grant_type`	`authorization_code` or `refresh_token`
`code` + `code_verifier`	From the callback and step 1
Client authentication	Secret or private-key JWT, if confidential

The token endpoint returns an access token, often a refresh token, and an ID token if you asked for `openid`. Confidential clients (servers that can keep a secret) authenticate here; public clients (SPAs, mobile) cannot. A refresh token for a public client must be sender-constrained (DPoP, mTLS) or rotated on every use.

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic d2ViLWFwcDpzM2NyM3Q=

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r
```

> **Gotcha:** OAuth 2.1 drops `redirect_uri` from this request, but OAuth 2.0 and OpenID Connect servers still require it when it was in the authorize request. A 2.1-only client that omits it gets an error — keep sending it until your provider documents 2.1 behavior.

Client credentials and device flow

Grant	Use for	User involved?
`client_credentials`	Service-to-service calls	No
Device code (RFC 8628)	TVs, CLIs, no browser	Yes, on another device
Authorization code	Everything with a browser	Yes

Client credentials means the client acts as itself, so only a confidential client may use it, and there is no user and no refresh token. The device flow shows the user a short code to enter on a phone or laptop while the device polls the token endpoint.

```
GT=urn:ietf:params:oauth:grant-type
curl -s https://as.example.com/token \
  -d "grant_type=$GT:device_code" \
  -d device_code=GmRhmhcxhwAzkoEq \
```

```
  -d client_id=tv-app
# {"error":"authorization_pending"} until done
```

> **Note:** While the user has not finished, the device-code poll returns `authorization_pending`; on `slow_down`, add 5 seconds to your polling interval.

ID token vs access token

	ID token	Access token
Audience	The client (`client_id`)	The API
Read by	The client	The resource server
Format	Always a JWT	Opaque or JWT
Says	Who signed in, how, when	What the bearer may do

The ID token is OIDC's addition: a signed JWT the client validates to learn the user's identity. The stable user key is the pair `iss` + `sub`, never the email. The access token

is opaque to the client — it may change format at any time, so never parse it in the client.

```
{
  "iss": "https://as.example.com",
  "sub": "248289761001",
  "aud": "web-app",
  "exp": 1793635200,
  "iat": 1793634600,
  "nonce": "n-0S6_WzA2Mj",
  "auth_time": 1793634590
}
```

> **Warning:** Never send an ID token to an API as a credential. Its audience is your client, so an API that accepts it lets any app holding a user's ID token for that client call it.

Discovery, JWKS and validation

Check	ID token (client)	JWT access token (API)
Signature via `jwks_uri` key	Yes	Yes
`iss` exact match	Yes	Yes
`aud` contains	Your `client_id`	Your API identifier
`exp` in the future	Yes	Yes
Extra	`nonce` matches	`typ` is `at+jwt`, scopes

Fetch the provider's metadata once from `/.well-known/openid-configuration` (OAuth-only servers: `/.well-known/oauth-authorization-server`). It lists the endpoints and the `jwks_uri`. Cache the key set and refetch when a token arrives with an unknown `kid`, which is how key rotation works.

```
AS=https://as.example.com
curl -s "$AS/.well-known/openid-configuration" \
  | jq '{issuer, token_endpoint, jwks_uri}'
```

> **Gotcha:** Pin the accepted signing algorithms in code. A validator that trusts the token's own `alg` header accepts `none` or an HMAC token signed with your public key as the secret.

Browser apps: the BFF pattern

Pattern	Tokens in the browser?
Backend for Frontend (BFF)	None — session cookie only
Token-mediating backend	Access token only
Browser-based OAuth client	All of them

RFC 10017 and the 2.1 draft rank these in decreasing order of security. In a BFF, your server is the confidential OAuth client: it runs the code flow, keeps the tokens server-side, and proxies API calls. The browser only ever holds an `HttpOnly` session cookie, so XSS cannot steal a token.

```
GET /bff/api/orders/42 HTTP/1.1
Host: app.example.com
Cookie: __Host-sid=k3Jx...

HTTP/1.1 200 OK
Content-Type: application/json
```

> **Tip:** A BFF turns your SPA into a cookie-session app again, so the cookie and CSRF rules in the web authentication sheet apply to it.

Common misconfigurations

Mistake	Fix
Wildcard or prefix redirect URIs	Register exact URIs
Long-lived access tokens	Minutes, plus refresh
Broad scopes “to be safe”	Request the least you need
Secret embedded in an SPA or app	It is a public client — use PKCE
Logging tokens or full URLs	Redact `code`, tokens

Most real breaches are one of these, not a broken spec. Redirect URI matching is the big one: any loosening lets an attacker register a URI that receives the victim’s code.

```
Registered:  https://app.example.com/cb
Accepted:    https://app.example.com/cb
Rejected:    https://app.example.com/cb/
```

```
Rejected:    https://app.example.com/cb?x=1
Rejected:    https://evil.app.example.com/cb
```

> **Note:** Native apps use a claimed `https` link or a loopback redirect (`http://127.0.0.1:<port>`) per RFC 8252; the AS must allow any port for loopback.

Further reading

- OAuth 2.1 Internet-Draft
- RFC 9700 — OAuth 2.0 Security Best Current Practice
- RFC 10017 — OAuth 2.0 for Browser-Based Applications
- RFC 8628 — Device Authorization Grant
- OpenID Connect Core 1.0
- OpenID Connect Discovery 1.0