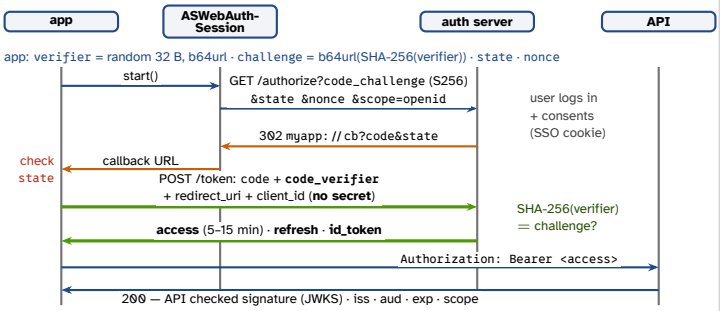
OAuth 2.0 · OIDC · JWT — for a native app

In one line: OAuth 2.0 (RFC 6749) is *delegated authorization*: the app gets a scoped **access token** without ever seeing the password. **OIDC** adds *authentication*: an **ID token** saying who logged in. A native app is a **public client** (no secret), so it uses **authorization code + PKCE** (RFC 7636) in the **system browser** (RFC 8252). A **JWT** (RFC 7519) is *signed, not encrypted*.

Authorization code + PKCE (iOS)



PKCE stops: another app claiming `myapp://` steals the `code` — without the `verifier`, `/token` refuses it.

JWT = header . payload . signature

eyJhbGciOiJSUzI1NiIs….eyJpc3MiOiJodHRwczov….SflKxwRJSMeKKF2QT4…

header
`"alg": "RS256"`
`"kid": "2025-09"`
`"typ": "JWT"`

payload (claims)
`"iss"` issuer URL `"sub"` user id
`"aud"` API / client_id
`"exp" "nbf" "iat"` seconds
`"jti"` unique id `"scope"`

signature = RS256(key `kid`, `b64url(header) + "." + b64url(payload)`)
verify with the issuer's public **JWKS** (`jwks_uri`, from `/.well-known/openid-configuration`)

base64url, no padding → **anyone can read the payload**: no secrets, no PII beyond need. JWE encrypts; almost every "JWT" is JWS.

3 tokens: **access** → the API (aud = API; app treats it as opaque) · **refresh** → only the auth server · **ID token** → the *app* (aud = client_id) — never a bearer for the API.

How it works

- **Roles**: resource owner (user) · client (app) · authorization server (issues tokens) · resource server (API). **Scopes** cap what a token may ever do; the API still does per-user authZ.
- **Implicit is dead**: token in the redirect *fragment* — leaks via history/referrer/logs, no refresh, nothing to bind PKCE to. OAuth 2.1 and the Security BCP (RFC 9700) drop it, and the password grant. Machine-to-machine: client credentials.
- **Redirect**: `ASWebAuthenticationSession(url:callbackURLScheme:)` — the app never sees the password, shares Safari's SSO cookies; `prefersEphemeralWebBrowserSession` opts out. Claimed `https` redirect (universal link) beats a custom scheme. **Not** a `WKWebView`: the app could read the password; Google refuses embedded views.
- **Refresh rotation**: every refresh returns a *new* RT and kills the old; an old RT presented again = theft → the server revokes the whole **family**. Alternative: sender-constrained tokens (DPoP, RFC 9449).
- **OIDC**: `scope=openid`; ID token claims `iss sub aud exp iat nonce auth_time`; `/userinfo` takes the access token. Identity key = `iss + sub`, never email. Check the `nonce` you sent — binds the token to this login (replay).

JWT validation — server side, every request

1. `alg` in **your allow-list** (e.g. only RS256/ES256) — never `none`; never let the token's `alg` pick the key type (RS256→HS256 confusion signs with the *public* key).
2. `kid` → key from cached JWKS; unknown `kid` → refetch once (rate-limited).
3. Verify the signature over the bytes *as received*.
4. `iss` exact · aud contains *me* · `exp` > now − leeway, `nbf` ≤ now + leeway (30–60 s skew) · `scope` · `jti` if one-time.

ID token from `/token` over TLS: OIDC lets the *app* skip the signature check, but still check `iss aud exp nonce`.

Example — single-flight refresh

```
actor TokenStore {
  private var access: Token                    // in memory
  private var refreshing: Task<Token, Error>?
  func valid() async throws -> Token {
    if let r = refreshing { return try await r.value } // join
    guard access.expiresWithin(60) else { return access }
    let r = Task { try await auth.refresh(keychain.refreshToken) }
    refreshing = r                    // set BEFORE the await
    defer { refreshing = nil }
    access = try await r.value        // new RT saved to Keychain
    return access                     // 401 later? same path, forced
  }
}
```

Tokens at a glance

	for	life	on iOS
access	API	5–15 min	memory (+ Keychain)
refresh	auth server	days, rotated	Keychain only
ID	the app	minutes	read claims once, then drop

Storing tokens on iOS

`SecItemAdd` with `kSecClassGenericPassword`. Accessibility: `AfterFirstUnlockThisDeviceOnly` if a background refresh must read it while locked; `WhenUnlockedThisDeviceOnly` otherwise. `ThisDeviceOnly` = no backup/migration to a new phone. Biometry (`SecAccessControl .biometryCurrentSet`) on the RT blocks *silent* refresh — a deliberate trade. Never `UserDefaults`. Keychain items **survive uninstall** (undocumented) → wipe on first launch after reinstall.

Logout and revocation — the JWT limit

A JWT is valid until `exp`: nothing *un-signs* it. Levers: short access TTL · revoke the RT (`/revoke`, RFC 7009) · deny-list by `jti` or a per-user token version (state again) · opaque tokens + introspection (RFC 7662). Logout = revoke RT + delete Keychain + end the browser session (`end_session_endpoint`) — or the next login is silent.

Interview traps

- ! Two requests hit 401, two refreshes run with rotation: the second uses a dead RT → reuse detection revokes the family → "random logouts". Single-flight it.
- ! Sending the **ID token** to your API (wrong aud) or reading authZ from it.
- ! A client secret in the binary is public — extractable. PKCE replaces it.
- ! `exp` is NumericDate **seconds**, not ms; compare with a leeway.
- ! OAuth alone is not login: an access token says *what*, not *who*.
- ! Sign in with Apple returns name/email only on the **first** authorization — store them.

Remember

"Code + verifier in the browser; access to the API, ID to the app, refresh to nobody else; check alg, kid, iss, aud, exp."

Likely questions

1. What does PKCE bind? — the code to the app that asked.
2. Why not implicit? — token in URL, no refresh, no binding.
3. Instant JWT logout? — no: short TTL + revoke RT + deny-list.
4. Is a JWT encrypted? — no: base64url, signed only.

Related: tls-pki (mTLS instead of bearer) · keychain-secure-enclave · swift6-strict-concurrency (actor reentrancy) · api-design (401/403) · deep-links-universal-links (https redirect)