17 · Authentication

Flutter Practice Notes

Contents

17 · Authentication

Auth Fundamentals (AuthN vs AuthZ, Session Models)

Token-Based Auth (JWT, Access/Refresh, Auto-Refresh)

OAuth 2.0 & Social Login (PKCE, Google/Apple)

Biometric Authentication (`local_auth`)

Session Management (Storage, Guards, Logout, Lifecycle)

17 · Authentication

Introduction

Authentication proves *who* a user is and maintains their *session*. This module covers auth fundamentals, token-based auth (JWT + access/refresh), OAuth 2.0 / social login, biometrics, and secure session management — integrating secure storage (15), interceptors (16), and route guards (13).

Why this module exists

Auth is security-critical and error-prone: tokens leaking to prefs/logs, no refresh flow, sessions that don't clear on logout, insecure OAuth. Getting it right protects users and is a frequent interview/architecture topic.

Module map

#	File	Topic	Level
1	01_auth_fundamentals.md	AuthN vs AuthZ, session models	🔵
2	02_token_auth_jwt.md	JWT, access/refresh, auto-refresh	🔴
3	03_oauth_and_social_login.md	OAuth 2.0, PKCE, Google/Apple sign-in	🔴
4	04_biometric_auth.md	`local_auth`, fingerprint/face	🔵
5	05_session_management.md	Token storage, guards, logout, lifecycle	🔴

> **Cross-references:** Secure storage: 15 · secure_storage. Interceptors/refresh: 16 · rest_client_and_interceptors. Route guards: 13 · guards_and_redirects. Firebase Auth: Module 18. Security/threat model: Module 37. State (auth state): Module 11.

Prerequisites

15 Local Storage, 16 Networking, 13 Routing.

What you'll be able to do after this module

- Distinguish authentication from authorization and choose a session model.
- Implement JWT access/refresh with an auto-refresh interceptor.
- Integrate OAuth 2.0 / social login securely (PKCE).
- Add biometric authentication.

- Manage sessions end-to-end: secure storage, guards, logout, expiry.

Capstone

Secure auth flow: Login → store tokens in secure storage → auto-attach + refresh via interceptor → guard routes by auth state → biometric unlock → logout that clears everything — a production-grade session backbone.

Summary

Authentication is a security-critical pipeline: prove identity, issue/refresh tokens, store them securely, guard access, and clear on logout. Build it once, correctly, behind an auth repository the app depends on.

Auth Fundamentals (AuthN vs AuthZ, Session Models)

> **Authentication (AuthN)** proves *who you are*; **authorization (AuthZ)** decides *what you can do* — and a session model (stateful server sessions vs stateless tokens) determines how identity persists across requests.

Introduction

Before implementing login, get the concepts straight: authentication vs authorization, and the two session models (server-side sessions with cookies vs stateless tokens like JWT). These decisions shape the whole auth architecture the later files build on.

Why this concept exists

Confusing AuthN and AuthZ leads to security holes (authenticated ≠ allowed). And choosing the wrong session model (or mixing them) causes scaling, security, and UX problems. Clear fundamentals prevent both.

Real-world analogy

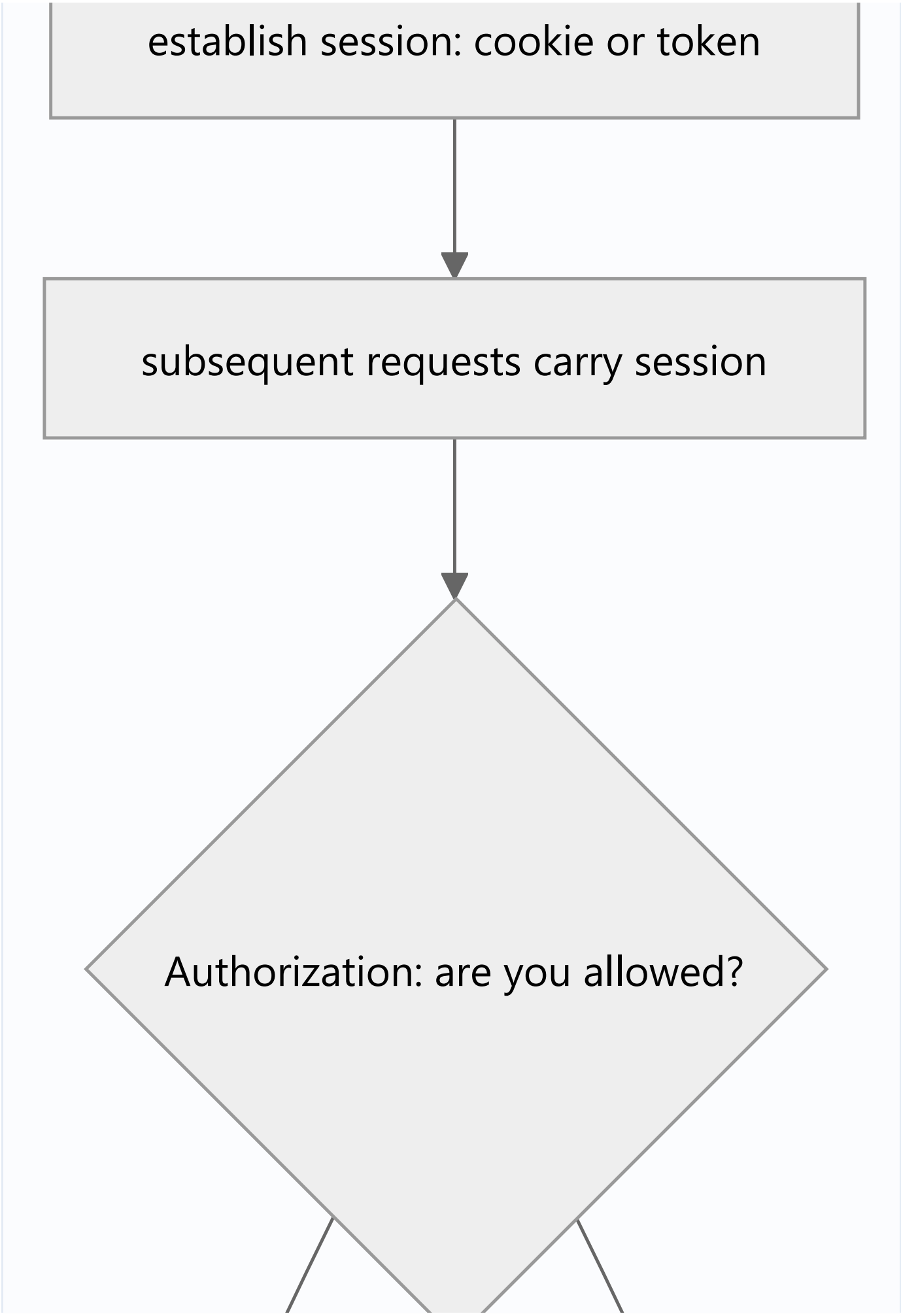
AuthN = showing your **ID at the door** (proving identity). **AuthZ** = your **ticket/badge** deciding which rooms you may enter (VIP vs general). A **session** is the **wristband** you wear after entry so you don't re-show ID at every room — a stamped wristband (server session) vs a self-verifying hologram badge (stateless token).

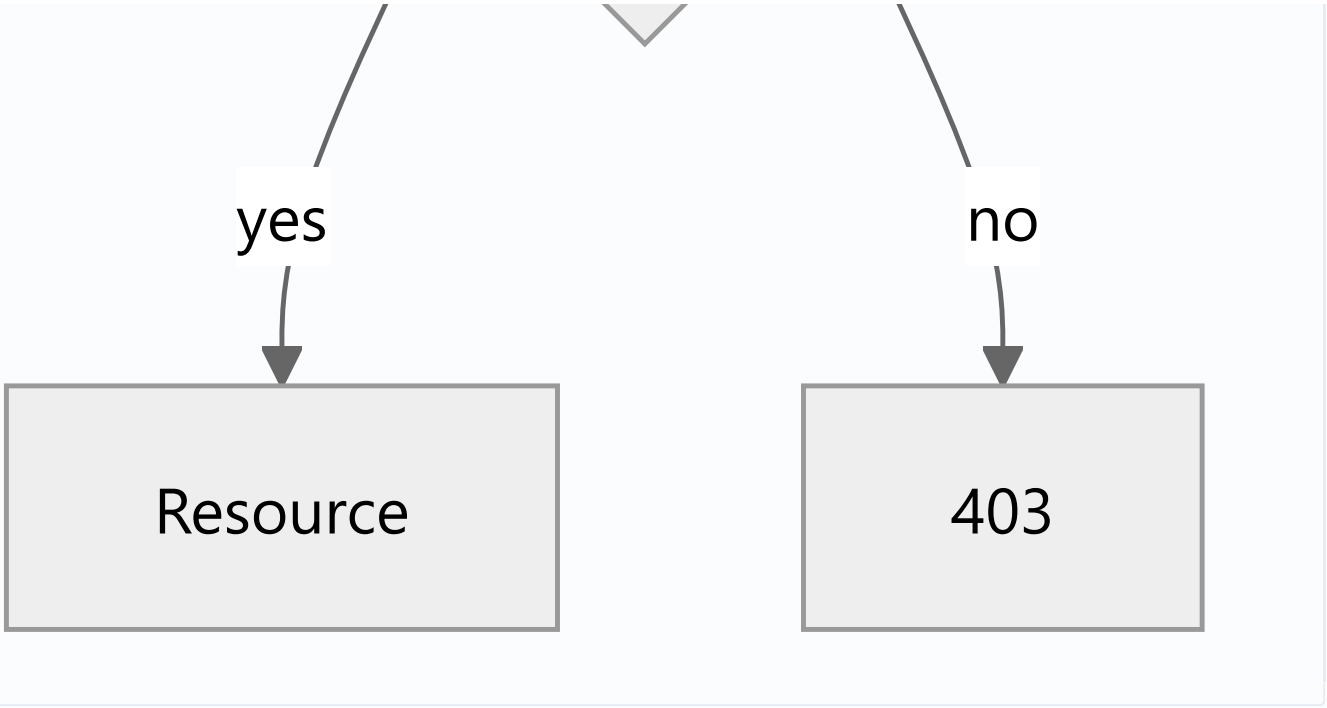
Problem Statement

A user logs in and accesses an admin page. Which step verifies identity, which verifies permission, and how does the app "remember" them across requests? You'll separate AuthN/AuthZ and pick a session model.

Internal Working


credentials
Authentication: who are you?
verified


establish session: cookie or token
subsequent requests carry session
Authorization: are you allowed?



- **Authentication (AuthN)**: verify identity (password, OTP, biometric, OAuth). Result: a session/token.
- **Authorization (AuthZ)**: given identity, check permissions/roles/scopes for a resource. `401` = not authenticated; `403` = authenticated but not authorized.
- **Session models**:
  - **Stateful (server-side sessions)**: server stores session state; client holds a session **cookie** (session id). Easy to revoke; needs server memory/sticky sessions; CSRF considerations.
  - **Stateless (tokens/JWT)**: the token itself carries claims and is verified by signature — server stores nothing. Scales horizontally; harder to revoke (needs short expiry + refresh/blacklist). Common in mobile (02_token_auth_jwt.md).
- Mobile apps typically use **stateless tokens** (bearer tokens in the `Authorization` header) rather than cookies.

Memory Representation

Sessions/tokens are stored client-side: tokens in **secure storage** (never prefs — 15 · secure_storage); server session state (stateful) lives on the server.

Compiler Behavior / Runtime Behavior

Not applicable directly; auth state is often modeled as a `Listenable`/stream driving guards (13 · guards_and_redirects).

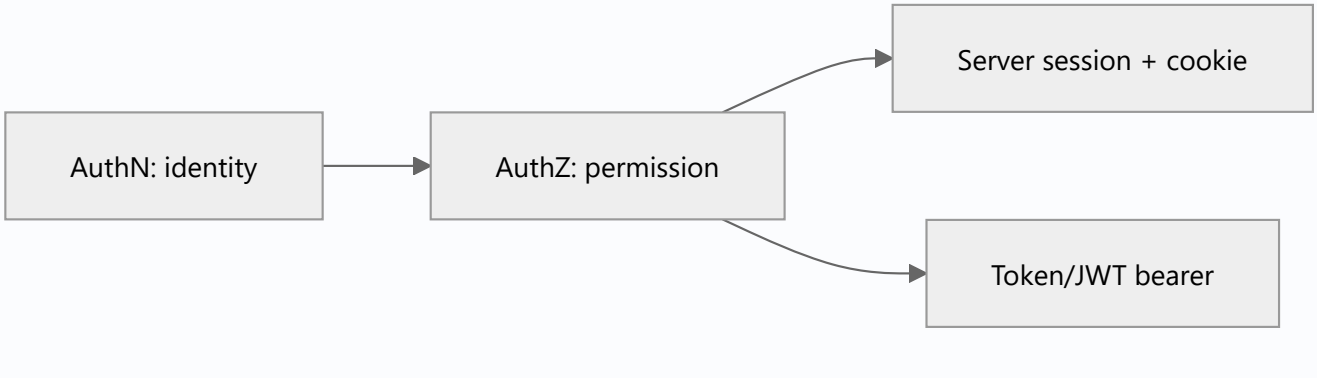
Flutter Engine Behavior / Dart VM Behavior

Not applicable.

Examples

```
// Modeling auth state (drives guards + UI)
sealed class AuthState { const AuthState(); }
class Authenticated extends AuthState { final User user; const Authenticated(this.user); }
class Unauthenticated extends AuthState { const Unauthenticated(); }
class AuthLoading extends AuthState { const AuthLoading(); }

class User {
  final String id;
  final Set<String> roles; // for AuthZ (e.g., {'admin'})
  const User(this.id, this.roles);
  bool can(String role) => roles.contains(role); // authorization check
}

// AuthN: verify identity -> session. AuthZ: check permission.
bool canAccessAdmin(AuthState state) => switch (state) {
      Authenticated(:final user) => user.can('admin'), // authenticated AND authorized
      _ => false,                                      // 401 vs 403 handled upstream
    };
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Treating authenticated as authorized	Security hole	Check permissions/roles separately (AuthZ)
Confusing 401 vs 403	Wrong handling/UX	401 = re-authenticate; 403 = show "no access"
Tokens in `SharedPreferences`	Plain text leak	Secure storage (15)
Client-only authorization	Bypassable	Enforce AuthZ server-side; client checks are UX only
Mixing session models inconsistently	Confusion/bugs	Pick one (tokens for mobile)

Best Practices

- Separate **AuthN** (identity) from **AuthZ** (permissions); handle **401 vs 403** distinctly.
- For mobile, prefer **stateless tokens** (bearer) with short expiry + refresh (02_token_auth_jwt.md).
- Store tokens in **secure storage**; never trust client-side authorization for security (server enforces).
- Model **auth state** as a `Listenable`/stream to drive route guards and UI (13).
- Front auth behind an `AuthRepository` the app depends on.

Performance

Stateless tokens avoid server session lookups (scale better); short-lived tokens + refresh balance security and UX. Negligible client cost (Module 21).

Advantages / Disadvantages

- **+ Stateless tokens:** scalable, mobile-friendly, self-contained. **+ Stateful sessions:** easy revocation, server control.
- **− Stateless:** revocation is hard (short expiry/blacklist). **− Stateful:** server state, sticky sessions, CSRF.

Interview Questions

1. 🟢 **Authentication vs authorization?** — AuthN proves identity (who you are); AuthZ decides permissions (what you can do).
2. 🟢 **401 vs 403?** — 401 Unauthorized = not authenticated (log in); 403 Forbidden = authenticated but not permitted.
3. 🟡 **Stateful sessions vs stateless tokens?** — Server stores session state (cookie holds an id; easy revoke, needs server state) vs self-verifying tokens (no server state; scalable; harder to

revoke).

4. 🟡 **Which session model do mobile apps typically use?** — Stateless bearer tokens (JWT) in the `Authorization` header — not cookies.
5. 🟡 **Where should tokens be stored on-device?** — Secure storage (Keychain/Keystore), never `SharedPreferences`.
6. 🔴 **Why is client-side authorization insufficient?** — Clients can be tampered with; authorization must be enforced server-side (client checks only improve UX).
7. 🔴 **What's the revocation tradeoff with stateless tokens?** — They can't be easily invalidated before expiry; mitigate with short-lived access tokens + refresh and/or a server-side blacklist.

Senior Engineer Tips

- Always ask "authenticated *and* authorized?" — model roles/scopes explicitly and enforce them server-side.
- Represent auth as a sealed state (`Authenticated` / `Unauthenticated` / `Loading`) so guards and UI are exhaustive and reactive.
- For mobile, standardize on short-lived access tokens + refresh in secure storage from day one.

Architect Perspective

The AuthN/AuthZ split and session-model choice are foundational security-architecture decisions. Stateless tokens + secure storage + reactive auth state + server-enforced authorization form the backbone that guards (13), interceptors (16), and the auth repository build on (Module 37).

Summary

- AuthN = identity; AuthZ = permission; 401 vs 403 differ accordingly.
- Stateful sessions (cookie) vs stateless tokens (JWT); mobile prefers stateless bearer tokens.
- Store tokens securely, enforce AuthZ server-side, model auth as reactive state behind a repository.

Revision Notes

- AuthN (who) vs AuthZ (what); 401 (re-auth) vs 403 (forbidden).
- Stateful (server session + cookie, revocable) vs stateless (JWT, scalable, hard revoke).
- Mobile → bearer tokens in secure storage; server enforces AuthZ.
- Model auth as sealed state → drives guards/UI; front with `AuthRepository` .

Practice Questions

1. Give an example where a user is authenticated but not authorized.
2. Why do mobile apps favor stateless tokens?
3. Why can't the client be the source of truth for authorization?

Coding Questions

1. Model a sealed `AuthState` + `User` with roles and a `can(role)` check.
2. Write a function distinguishing 401 vs 403 handling.
3. Sketch an `AuthRepository` interface (login/logout/currentUser).

Mini Project

Auth model (Dart): Define `AuthState` (sealed), a `User` with roles, and an `AuthRepository` interface; implement a fake that logs in with roles and expose auth state as a `Stream` / `Listenable`. Write a permission check (`can`). Acceptance: AuthN/AuthZ separated; reactive auth state; roles enforced in the model; analyzer clean.

Token-Based Auth (JWT, Access/Refresh, Auto-Refresh)

> A **JWT** is a signed, self-contained token carrying claims (user id, roles, expiry); apps use a short-lived **access token** for requests plus a longer-lived **refresh token** to obtain new access tokens — with an interceptor that auto-refreshes on 401 so the user never sees it.

Introduction

The dominant mobile auth pattern: login returns an **access token** (short expiry) + **refresh token** (longer). Requests carry the access token; when it expires, a refresh flow silently exchanges the refresh token for a new access token. This file covers JWT structure, the access/refresh model, and the auto-refresh interceptor.

Why this concept exists

Long-lived access tokens are dangerous if leaked; very short ones would force constant re-login. The access+refresh split gives **short-lived risk exposure** for access tokens plus a **seamless UX** via refresh — the practical balance for stateless mobile auth (01_auth_fundamentals.md).

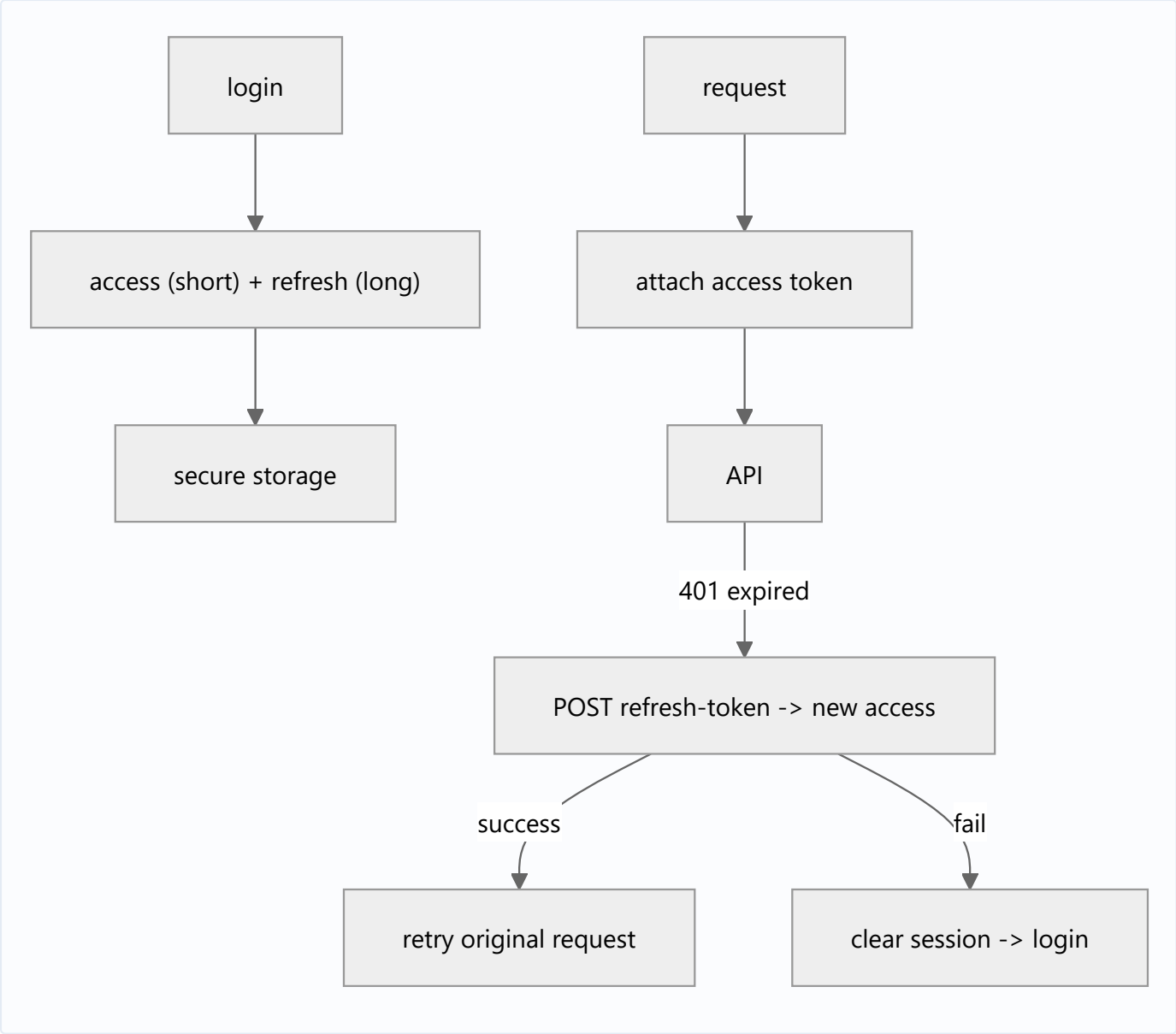
Real-world analogy

An **event wristband that expires hourly** (access token) plus a **membership card** (refresh token): when the wristband expires, you quietly show your membership card at a kiosk to get a fresh wristband — without leaving the event or re-registering.

Problem Statement

Login yields access+refresh tokens; requests attach the access token; when it expires (401), the app must refresh and retry transparently, and force re-login if refresh fails. You'll implement the model + auto-refresh interceptor.

Internal Working



- **JWT structure**: `header.payload.signature` (base64url) — payload holds **claims** (`sub` , `exp` , `roles` , etc.); the signature (server's secret/key) verifies integrity. **Never trust an unverified JWT client-side for security; the server verifies.** You may read `exp` /claims client-side for UX, but don't rely on it for authorization.
- **Access token**: short-lived (minutes), sent as `Authorization: Bearer` . Limits damage if leaked.
- **Refresh token**: longer-lived, stored securely, used only to get new access tokens (often rotated on use).
- **Auto-refresh flow** (interceptor — 16 · rest_client_and_interceptors): on a 401, call the refresh endpoint, save the new access token, retry the original request **once**; on refresh failure, clear the session and route to login. Guard against **concurrent refreshes** (queue in-flight requests) and **infinite loops**.
- **Storage**: both tokens in **secure storage**, never prefs/logs (15 · secure_storage).

Memory Representation

Tokens are small strings held briefly in memory when read; persisted encrypted in secure storage. Don't log them (Module 37).

Compiler Behavior

Not applicable.

Runtime Behavior

Each request attaches the access token; a 401 triggers refresh+retry; simultaneous 401s must share one refresh (lock/queue) to avoid multiple refreshes. Refresh failure → logout.

Flutter Engine Behavior / Dart VM Behavior

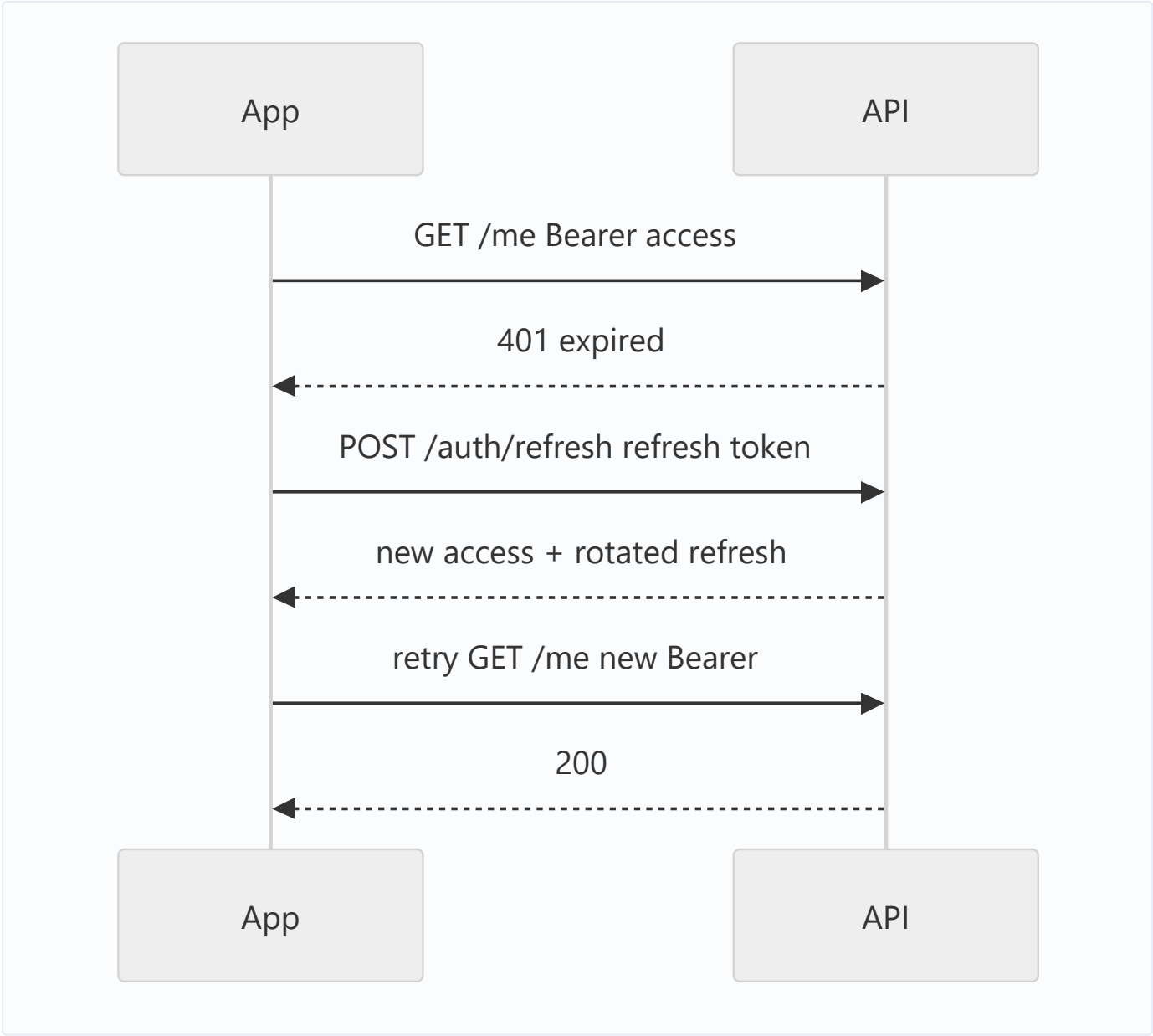
Not applicable.

Examples

```
import 'package:dio/dio.dart';

abstract interface class TokenStore {          // backed by secure storage (Module 15)
  Future<String?> get accessToken;
  Future<String?> get refreshToken;
  Future<void> save({required String access, required String refresh});
  Future<void> clear();
}

class AuthInterceptor extends Interceptor {
  final Dio dio;                 // a bare dio for refresh (no interceptor loop)
  final TokenStore store;
  final Future<void> Function() onSessionExpired; // e.g., route to login
  bool _refreshing = false;
  AuthInterceptor(this.dio, this.store, this.onSessionExpired);

  @override
  void onRequest(RequestOptions options, RequestInterceptorHandler handler) async {
    final token = await store.accessToken;
    if (token != null) options.headers['Authorization'] = 'Bearer $token';
    handler.next(options);
```

```
  }

  @override
  void onError(DioException err, ErrorInterceptorHandler handler) async {
    // Only handle 401, and avoid refreshing the refresh call itself:
    final is401 = err.response?.statusCode == 401;
    final isRefreshCall = err.requestOptions.path.contains('/refresh');
    if (!is401 || isRefreshCall) return handler.next(err);

    try {
      final newAccess = await _refresh();      // shared/guarded refresh
      // retry original request with the new token (once):
      final req = err.requestOptions..headers['Authorization'] = 'Bearer $newAccess';
      final res = await dio.fetch(req);
      return handler.resolve(res);
    } catch (_) {
      await store.clear();
      await onSessionExpired();                // force re-login
      return handler.next(err);
    }
  }

  Future<String> _refresh() async {
    // (guard concurrent refreshes with a lock/queue in production)
    final refresh = await store.refreshToken;
    if (refresh == null) throw StateError('no refresh token');
    final res = await dio.post('/auth/refresh', data: {'refresh_token': refresh});
    final access = res.data['access_token'] as String;
    final newRefresh = res.data['refresh_token'] as String; // rotation
    await store.save(access: access, refresh: newRefresh);
    return access;
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Long-lived access tokens	Big damage if leaked	Short access + refresh
Refreshing on every 401 without guarding concurrency	Multiple refreshes / race	Single in-flight refresh (lock/queue)
Refresh loop (refresh call 401s)	Infinite loop	Exclude the refresh endpoint from the interceptor
Tokens in prefs/logs	Leak	Secure storage; never log tokens
Trusting client-decoded JWT for AuthZ	Bypassable	Server verifies; client claims are UX only
Not clearing tokens on refresh failure	Stuck/insecure session	Clear + route to login

Best Practices

- Use **short-lived access + longer refresh**; **rotate** refresh tokens on use.
- Auto-refresh in an **interceptor**: on 401 → refresh → retry once; exclude the refresh call; **serialize concurrent refreshes**.
- Store both tokens in **secure storage**; never log them.
- On refresh failure, **clear the session** and route to login (drive via reactive auth state + guards — 13).
- Treat client-decoded claims as **UX-only**; the server is the authority.

Performance

Refresh happens rarely (only on expiry); serialized refresh avoids redundant calls. Negligible overhead; improves UX by hiding token expiry (Module 21).

Advantages / Disadvantages

- **+** Stateless/scalable, seamless UX (silent refresh), limited leak exposure (short access), server-verifiable.
- **−** Refresh flow complexity (concurrency/loops), revocation still hard (short expiry mitigates), must store securely.

Interview Questions

1. 🟢 **What is a JWT?** — A signed, self-contained token (`header.payload.signature`) carrying claims (user id, roles, expiry); the server verifies the signature.
2. 🟢 **Why access + refresh tokens?** — Short-lived access limits leak damage; a longer refresh token silently gets new access tokens for seamless UX.
3. 🟡 **How does auto-refresh work?** — An interceptor catches 401, calls the refresh endpoint, saves the new access token, and retries the original request once; on failure it forces re-login.
4. 🟡 **How do you avoid multiple concurrent refreshes?** — Serialize refresh (a lock/queue) so simultaneous 401s share one refresh call.
5. 🟡 **Where are tokens stored, and where verified?** — Stored in secure storage on-device; verified server-side (client-decoded claims are UX-only).
6. 🔴 **How do you prevent a refresh loop?** — Exclude the refresh endpoint from the auth interceptor and cap retries; on refresh 401, log out.
7. 🔴 **How is revocation handled with stateless JWTs?** — Short access expiry + refresh rotation, plus optional server-side refresh-token blacklist/rotation to invalidate stolen refresh tokens.

Senior Engineer Tips

- Get the **concurrency-guarded refresh** right early (queue in-flight requests, single refresh) — it's the #1 bug source.
- Rotate refresh tokens and detect reuse (a rotated token used twice = likely theft → revoke).
- Never log tokens; put refresh + storage behind an `AuthRepository` / `TokenStore` so the flow is centralized and testable.

Architect Perspective

The access/refresh JWT pattern with an auto-refresh interceptor is the standard stateless mobile auth backbone: it balances security (short exposure, rotation) and UX (silent refresh), integrates with secure storage, interceptors, guards, and reactive auth state, and keeps servers stateless/scalable (Modules 15, 16, 13, 37).

Summary

- JWT = signed claims; use short-lived access + longer refresh, both in secure storage.
- Auto-refresh via interceptor: 401 → refresh → retry once; guard concurrency + loops; logout on failure.
- Server verifies tokens; client claims are UX-only; rotate refresh tokens.

Revision Notes

- JWT: header.payload.signature; claims (`sub` / `exp` /roles); server verifies.
- Access (short, Bearer) + refresh (long, rotated) → secure storage; never log.
- Interceptor: 401 → refresh (exclude refresh call, serialize) → retry once; fail → clear + login.
- Client claims = UX only; revocation via short expiry + rotation/blacklist.

Practice Questions

1. Why short-lived access tokens?
2. How do you prevent concurrent-refresh races and refresh loops?
3. Why not trust a client-decoded JWT for authorization?

Coding Questions

1. Implement an `AuthInterceptor` doing 401→refresh→retry-once (excluding the refresh call).
2. Add a lock so concurrent 401s trigger a single refresh.
3. Force logout (clear tokens + route to login) on refresh failure.

Mini Project

JWT session flow (Flutter): Build a `TokenStore` (secure storage) + `AuthInterceptor` implementing access/refresh with concurrency-guarded auto-refresh, retry-once, refresh-call exclusion, and logout-on-failure; drive route guards from reactive auth state. Test 401→refresh→retry and refresh-failure→logout with a mocked `dio`. Acceptance: tokens secure; single guarded refresh; no loops; logout clears session; tests pass.

OAuth 2.0 & Social Login (PKCE, Google/Apple)

> OAuth 2.0 lets users grant your app limited access via a trusted provider (Google/Apple/etc.) without sharing their password; mobile apps use the **Authorization Code flow with PKCE** — an in-app browser redirect exchanges a code for tokens securely.

Introduction

Social login ("Sign in with Google/Apple") is OAuth 2.0/OpenID Connect in practice. This file covers the OAuth roles, the Authorization Code + **PKCE** flow (the secure mobile flow), and integrating Google/Apple sign-in — plus why implicit flow and embedded webviews are discouraged.

Why this concept exists

Users prefer not to create/manage yet another password; providers offer secure, trusted identity. OAuth delegates authentication to the provider and returns tokens/identity to your app — but doing it securely on mobile (no client secret, redirect handling) requires PKCE.

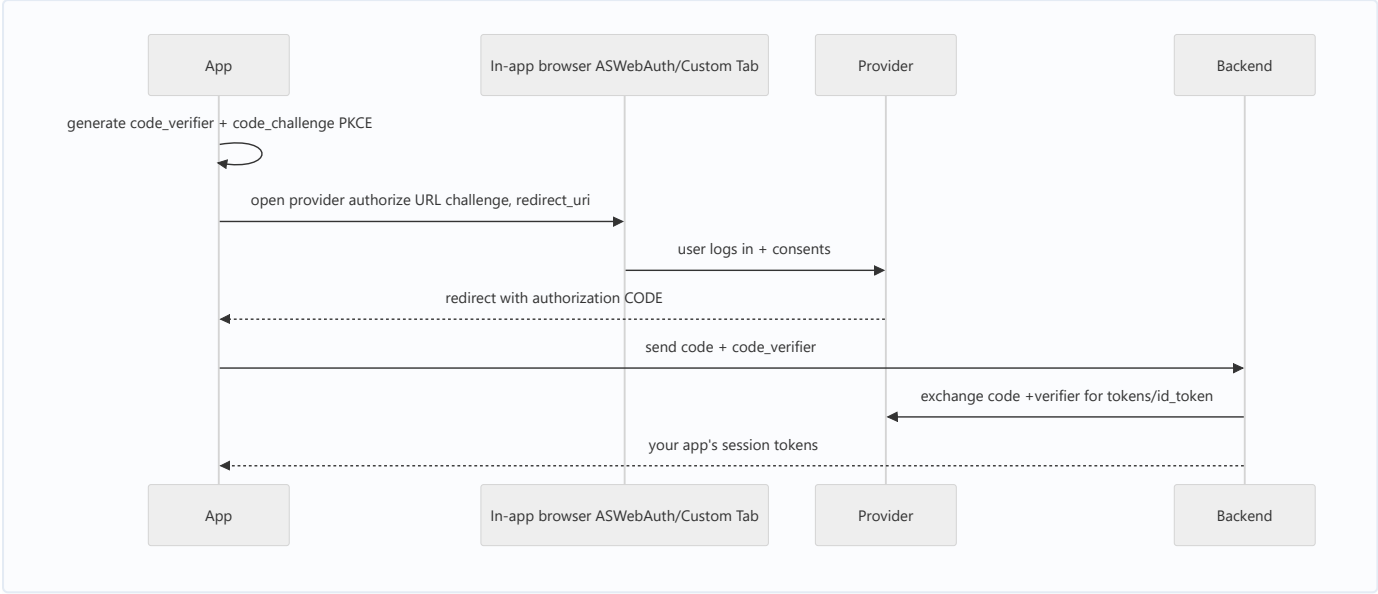
Real-world analogy

OAuth is a **valet key**: instead of handing over your master key (password), you give the valet (your app) a limited key (scoped token) issued by the car's manufacturer (provider) that only starts the car, not open the trunk. **PKCE** is a **one-time claim ticket** ensuring only the app that started the request can redeem the code.

Problem Statement

Add "Sign in with Google" and "Sign in with Apple," securely exchanging the provider's auth code for tokens/identity and establishing your app's session. You'll use the Authorization Code + PKCE flow via a sign-in package.

Internal Working



- **OAuth roles**: **Resource Owner** (user), **Client** (your app), **Authorization Server** (Google/Apple), **Resource Server** (API). **OIDC** adds an `id_token` (identity/claims) on top of OAuth (authorization).
- **Authorization Code + PKCE** (the mobile flow):
  1. App generates a `code_verifier` (random) + `code_challenge` (hash).
  2. Opens the provider's authorize URL in a **secure system browser** (ASWebAuthenticationSession / Android Custom Tabs), passing the challenge + `redirect_uri`.
  3. User authenticates/consents; provider redirects back with an **authorization code** (via a custom scheme / app link — 13 · deep_linking).
  4. App/backend exchanges code + `code_verifier` for tokens (PKCE proves the same app).
  5. Establish your app session (often your backend issues its own JWTs — 02_token_auth_jwt.md).
- **PKCE** replaces the client secret (mobile apps can't keep secrets) and prevents code interception.
- **Google/Apple**: use `google_sign_in` / `sign_in_with_apple` (or a generic `oauth2` / `AppAuth` / `flutter_appauth`); Apple sign-in is **required** by App Store if you offer other social logins.
- **Avoid**: the deprecated **implicit flow** and **embedded webviews** (phishing/credential risk) — use the system browser.

Memory Representation

Resulting provider/app tokens go in **secure storage** (15 · secure_storage); the `code_verifier` is short-lived in memory. Never log tokens/codes (Module 37).

Compiler Behavior / Runtime Behavior

The redirect returns to the app via a registered scheme/app link (platform config — 13 · deep_linking); the code exchange is a network call.

Flutter Engine Behavior

Sign-in packages open native secure browsers via the embedder and receive the redirect through platform deep-link config (10 · embedder_and_startup).

Dart VM Behavior

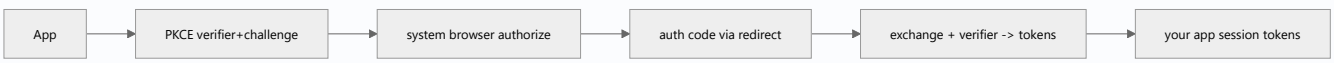
Not applicable.

Examples

```yaml
# pubspec.yaml
dependencies:
  google_sign_in: ^6.2.0
  sign_in_with_apple: ^6.1.0
```

```dart
import 'package:google_sign_in/google_sign_in.dart';
import 'package:sign_in_with_apple/sign_in_with_apple.dart';

class SocialAuth {
  final _google = GoogleSignIn(scopes: ['email']);

  // Google: package handles the OAuth flow; returns identity + tokens
  Future<String?> signInWithGoogle() async {
    final account = await _google.signIn();       // opens secure flow
    if (account == null) return null;             // user cancelled
    final auth = await account.authentication;     // idToken / accessToken
    // Send auth.idToken to YOUR backend -> backend verifies + issues your session:
    return auth.idToken;
  }

  // Apple: returns an authorization with identity token
  Future<String?> signInWithApple() async {
    final cred = await SignInWithApple.getAppleIDCredential(
      scopes: [AppleIDAuthorizationScopes.email, AppleIDAuthorizationScopes.fullName],
    );
    return cred.identityToken; // send to backend for verification
  }

  Future<void> signOut() => _google.signOut();
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Embedded webview for login	Phishing/credential risk, provider bans	Use system browser (ASWebAuth/Custom Tabs)
Implicit flow	Deprecated, insecure	Authorization Code + PKCE
Client secret in the app	Can't keep secrets on-device	PKCE (no secret)
Verifying `id_token` only on client	Spoofable	Verify server-side
Missing Apple sign-in when offering others	App Store rejection	Add Sign in with Apple
Redirect scheme/app link not configured	Callback never returns	Configure platform deep links (13)

Best Practices

- Use **Authorization Code + PKCE** via the **system browser**; never embedded webviews or implicit flow.
- **Verify tokens server-side**; have your backend issue your app's own session tokens (02_token_auth_jwt.md).
- Store resulting tokens in **secure storage**; never log codes/tokens.
- Offer **Sign in with Apple** if you offer other social logins (App Store requirement).
- Configure **redirect URIs** (custom scheme / app links) correctly per platform.
- Request **minimal scopes**; handle user cancellation gracefully.

Performance

Sign-in is an occasional, user-driven flow; negligible runtime cost. The main concerns are security and correct redirect handling, not performance.

Advantages / Disadvantages

- **+** No password to manage, trusted providers, better UX/conversion, standardized (OAuth/OIDC), secure with PKCE.
- **−** Redirect/deep-link + platform config complexity, provider dependency, Apple requirement, must verify server-side.

Interview Questions

1. 🟢 **What is OAuth 2.0?** — A delegation protocol letting users grant an app scoped access via a provider without sharing their password.

2. 🟢 **OAuth vs OpenID Connect?** — OAuth is authorization (access tokens/scopes); OIDC adds authentication/identity via an `id_token` on top.
3. 🟡 **Which OAuth flow do mobile apps use and why?** — Authorization Code with **PKCE** — secure without a client secret and resistant to code interception.
4. 🟡 **What is PKCE and what problem does it solve?** — Proof Key for Code Exchange: a verifier/challenge pair proving the same app that started the flow redeems the code — replaces the impossible-to-keep client secret on mobile.
5. 🟡 **Why avoid embedded webviews for login?** — Phishing risk (users can't verify the provider), credential exposure, and provider policy bans; use the system browser.
6. 🔴 **Why verify the provider token server-side?** — Client tokens can be spoofed; the backend must verify the `id_token`/code and issue your own session — client-only trust is insecure.
7. 🔴 **Why must you offer Sign in with Apple sometimes?** — App Store guidelines require it if the app offers other third-party social logins.

Senior Engineer Tips

- Let a maintained package handle the flow (`google_sign_in`, `sign_in_with_apple`, or `flutter_appauth` for generic OAuth); don't hand-roll redirect/PKCE.
- Always exchange/verify on the **backend** and issue your own session tokens — treat the provider result as an assertion to verify, not a session.
- Configure redirect schemes/app links carefully; test the callback on real devices (cold + warm start).

Architect Perspective

Social login delegates identity to trusted providers while your backend remains the session authority (verify → issue your own JWTs). Doing it via PKCE + system browser + server verification is the secure standard; it integrates with deep linking (callback), secure storage (tokens), and your session/guard layer (Modules 13, 15, 37).

Summary

- OAuth 2.0 (+OIDC) delegates auth to providers; mobile uses **Authorization Code + PKCE** via the system browser.
- Verify tokens server-side and issue your own session; store tokens securely; offer Apple sign-in when required.
- Avoid implicit flow/embedded webviews; configure redirects; request minimal scopes.

Revision Notes

- OAuth (authorization) + OIDC (`id_token` identity); roles: owner/client/authz server/resource server.
- Mobile flow: Authorization Code + **PKCE** (verifier/challenge, no client secret) via system browser; redirect via app link/scheme.
- Verify server-side → issue your own JWTs; tokens in secure storage; never log.
- Apple sign-in required if offering others; avoid implicit flow / embedded webviews.

Practice Questions

1. Why PKCE instead of a client secret on mobile?
2. Why use the system browser, not a webview?
3. Why verify the provider token on the backend?

Coding Questions

1. Implement Google + Apple sign-in returning identity tokens to send to a backend.
2. Handle user cancellation and errors gracefully.
3. Sketch the PKCE Authorization Code steps (verifier→challenge→code→exchange).

Mini Project

Social login (Flutter): Add Google + Apple sign-in that obtains a provider identity token, sends it to a (fake) backend to "verify and issue" your app's JWTs, stores them in secure storage, and updates reactive auth state. Handle cancel/errors; require Apple when Google is present. Acceptance: PKCE/system-browser flow via packages; server-verification step; tokens secure; cancellation handled; runs.

Biometric Authentication (`local_auth`)

> Biometric auth (fingerprint/Face ID) is a **local device unlock**, not a server login — use `local_auth` to gate access to an existing session or sensitive actions; the biometric result never leaves the device and must be backed by real credentials/tokens.

Introduction

`local_auth` (package) prompts the OS biometric/device-credential check and returns a simple pass/fail. This file covers correct usage (as a *local gate*, not identity to a server), availability checks, fallbacks, and the crucial distinction from network authentication.

Why this concept exists

Users want fast, secure re-entry ("unlock with Face ID") without re-typing passwords. Biometrics provide a convenient local factor. But biometrics **authenticate to the device**, not your server — they gate access to already-established, securely-stored credentials/sessions.

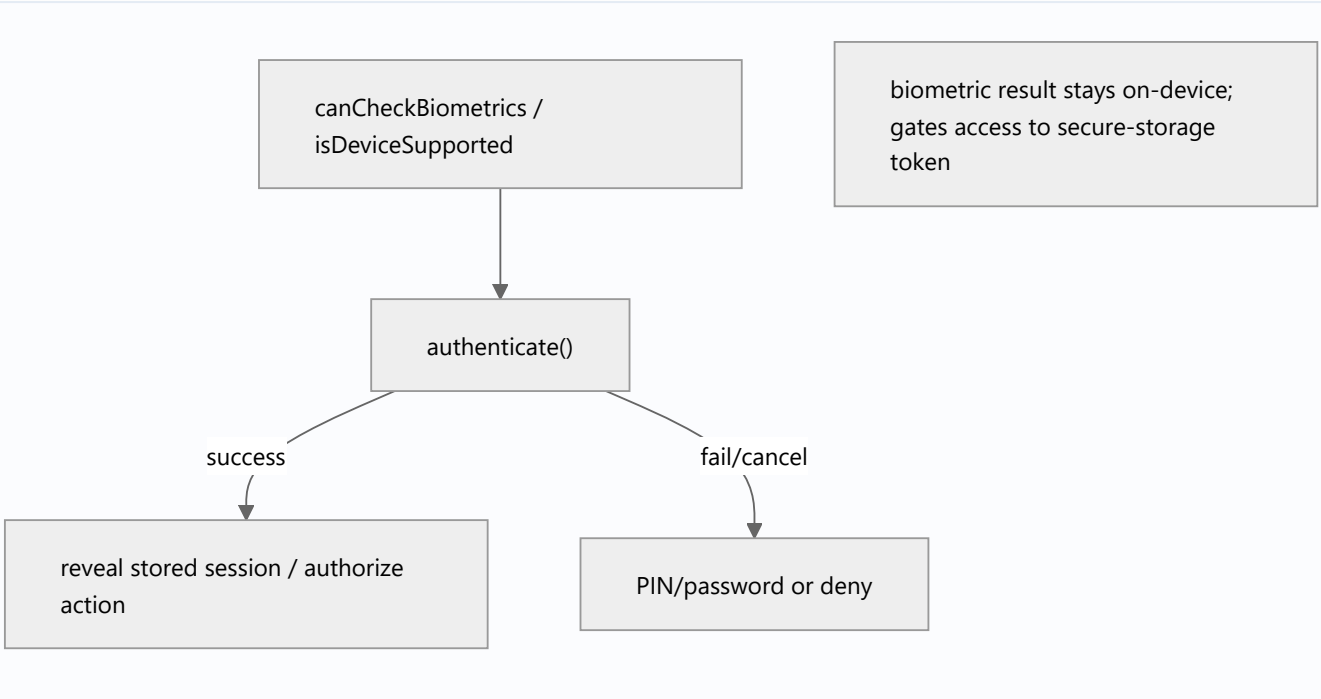
Real-world analogy

Biometrics are the **fingerprint lock on your phone case** protecting a wallet inside (your stored session token). Your fingerprint opens the case locally; it isn't shown to the bank. The bank still trusts the card (token) inside, not your fingerprint.

Problem Statement

After first login, let returning users unlock the app / authorize a payment with Face ID/fingerprint — falling back to PIN/password, handling unavailable hardware, and only unlocking a securely-stored session. You'll use `local_auth` correctly.

Internal Working



- **Availability**: `canCheckBiometrics`, `isDeviceSupported()`, `getAvailableBiometrics()` — check before prompting; handle no-hardware/not-enrolled.
- **Prompt**: `authenticate(localizedReason: ..., options: AuthenticationOptions(biometricOnly: false, stickyAuth: true))` returns `bool`. `biometricOnly: false` allows device PIN/passcode fallback.
- **Role**: a **local gate** — on success, reveal the app or read the securely-stored session token (15 · secure_storage) / authorize a sensitive action. It does **not** authenticate to your server.
- **Options**: `stickyAuth` (survive app backgrounding during prompt), error handling for lockout (too many attempts).
- **Security note**: biometrics are convenience over an existing session; the real credential is the stored token — protect it and require real login when the session is gone/expired.

Memory Representation

No biometric data is exposed to your app (OS-handled); you only get pass/fail. The gated secret lives in secure storage (15).

Compiler Behavior / Runtime Behavior

`authenticate()` is async; can throw platform exceptions (no hardware, not enrolled, locked out) — handle them. Result is a simple boolean.

Flutter Engine Behavior

`local_auth` calls native biometric APIs via the embedder (10 · embedder_and_startup); requires platform config (Face ID usage string on iOS, permissions on Android).

Dart VM Behavior

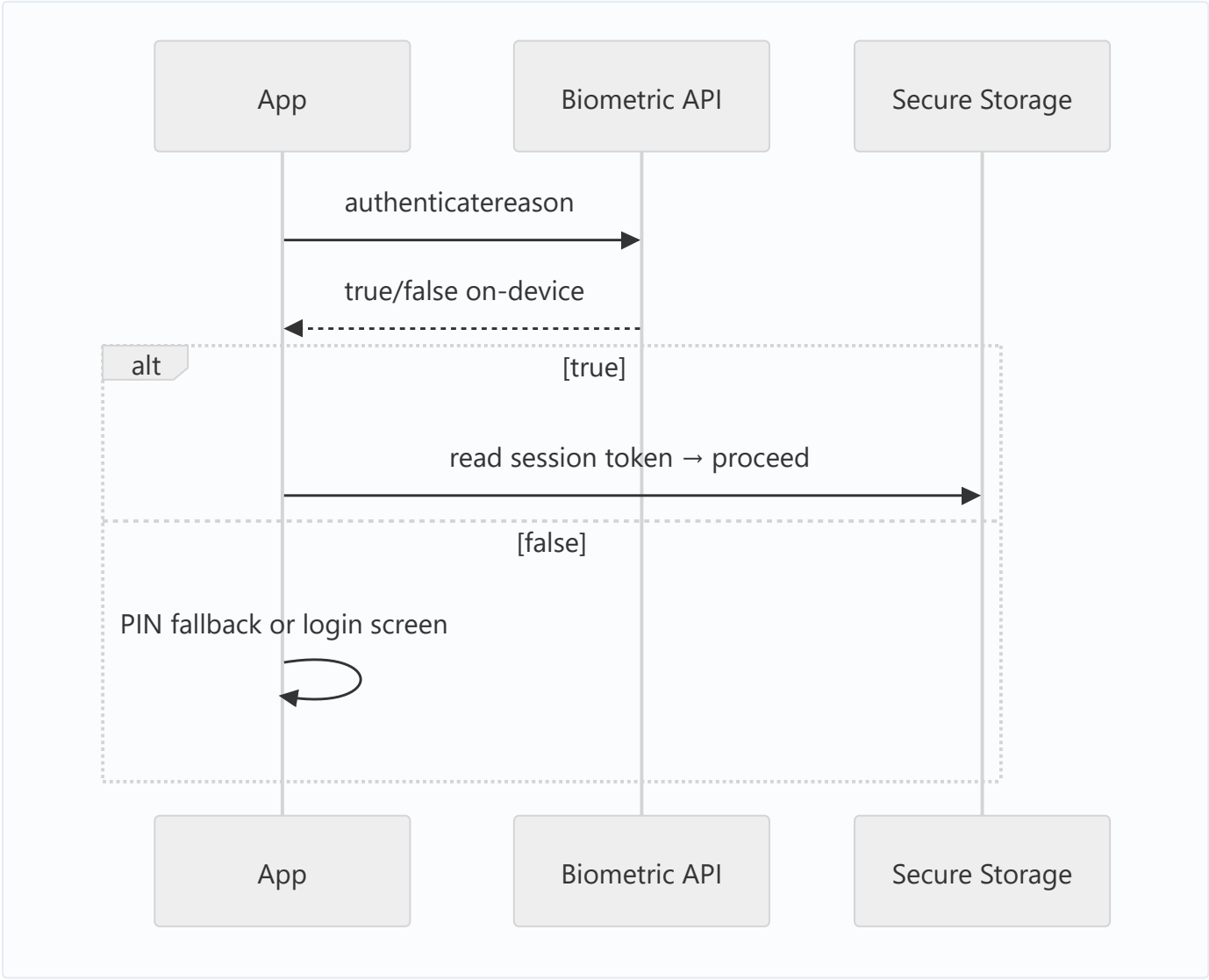
Not applicable.

Examples

```yaml
# pubspec.yaml
dependencies:
  local_auth: ^2.2.0
```

```dart
import 'package:local_auth/local_auth.dart';

class BiometricGate {
  final _auth = LocalAuthentication();

  Future<bool> isAvailable() async {
    try {
      return await _auth.isDeviceSupported() && await _auth.canCheckBiometrics;
    } catch (_) {
      return false;
    }
  }

  Future<bool> unlock() async {
    if (!await isAvailable()) return false; // fall back to password login
    try {
      return await _auth.authenticate(
        localizedReason: 'Unlock to access your account',
        options: const AuthenticationOptions(
          biometricOnly: false, // allow device PIN/passcode fallback
          stickyAuth: true,     // survive backgrounding during prompt
        ),
      );
    } on Exception {
      return false;             // not enrolled / locked out / cancelled
    }
  }
}

// Usage: on app resume, if a session token exists in secure storage,
// require BiometricGate.unlock() before revealing it; else go to login.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Treating biometrics as server auth	It's local-only, no server proof	Gate an existing token; server trusts the token
No availability check	Crash/poor UX on unsupported devices	Check `isDeviceSupported` / `canCheckBiometrics` first
No fallback	Locked out if biometrics fail	Allow PIN/password fallback
Storing a "biometric = logged in" flag insecurely	Bypassable	Gate the secure-storage token, not a bool
Ignoring lockout/enrollment errors	Crashes	Catch platform exceptions
Missing platform config (Face ID string)	Runtime failure/rejection	Add usage descriptions/permissions

Best Practices

- Use biometrics as a **local gate** over a securely-stored session/action — never as your only/server auth.
- **Check availability** and always provide a **PIN/password fallback** (`biometricOnly: false`).
- Handle errors (not enrolled, lockout, cancel) gracefully; use `stickyAuth` .
- Require **real login** when no valid session exists or it's expired.
- Configure platform requirements (iOS `NSFaceIDUsageDescription` , Android biometric permissions).

Performance

Instant, occasional, user-driven; negligible cost. It improves UX (fast re-entry) without network round-trips.

Advantages / Disadvantages

- + Fast, secure local re-entry/action authorization; no password typing; privacy-preserving (data stays on device).
- − Local-only (not server identity), device/enrollment dependent, needs fallback + platform config, can be bypassed if the underlying token isn't protected.

Interview Questions

1. 🟢 **What does biometric auth actually authenticate?** — The user *to the device* (local pass/fail); it does not authenticate to your server.
2. 🟢 **What should biometrics gate?** — Access to an already-established, securely-stored session/token or a sensitive action — not a fresh server login.
3. 🟡 **Why always provide a fallback?** — Hardware may be absent, not enrolled, or locked out; `biometricOnly: false` allows device PIN/passcode.
4. 🟡 **What checks precede a biometric prompt?** — `isDeviceSupported()` / `canCheckBiometrics` (and handle exceptions) before `authenticate()` .
5. 🟡 **Does biometric data reach your app?** — No; the OS handles it and returns only success/failure.
6. 🔴 **What's insecure about a "biometric passed → logged in" boolean?** — It's bypassable/tamperable; biometrics must gate the *actual* secure-stored token, and the server must still trust that token.
7. 🔴 **When must you force a full login despite biometrics?** — When there's no valid stored session or it's expired/revoked — biometrics can't create a session.

Senior Engineer Tips

- Frame biometrics as "unlock the wallet," not "prove identity to the bank" — the token is the credential; biometrics just gate local access to it.
- Always ship a fallback path; biometric-only flows strand users on unsupported/locked devices.
- Pair with app-lifecycle handling (re-lock on background) for sensitive apps (08 · app_lifecycle).

Architect Perspective

Biometrics are a UX-security convenience layer over the real session, not an identity provider. Architecturally they gate access to secure-stored tokens and sensitive actions, with mandatory fallback and platform config, complementing token auth and secure storage (02_token_auth_jwt.md, 15 · secure_storage, Module 37).

Summary

- Biometrics (`local_auth`) are a local device gate returning pass/fail — not server authentication.
- Gate access to a securely-stored session/action; always provide fallback; check availability; handle errors.
- The token is the real credential; force login when no valid session exists.

Revision Notes

- `local_auth`: `isDeviceSupported` / `canCheckBiometrics` → `authenticate(reason, options)` → bool.
- Local gate only (no server proof); gate secure-stored token/action; `biometricOnly:false` for PIN fallback; `stickyAuth`.
- Handle no-hardware/not-enrolled/lockout; platform config (Face ID string).
- Force login when session absent/expired; re-lock on background for sensitive apps.

Practice Questions

1. Why isn't biometric success a server login?
2. What must you check before prompting, and why a fallback?
3. What should biometrics actually protect/reveal?

Coding Questions

1. Build a `BiometricGate` with availability check, prompt, and fallback.
2. Gate reading a secure-storage token behind `unlock()`.
3. Handle not-enrolled/lockout exceptions gracefully.

Mini Project

Biometric unlock (Flutter): After first login (token in secure storage), gate app entry with `local_auth` (Face ID/fingerprint), falling back to PIN/password, checking availability, handling errors, and forcing full login when no valid session exists. Re-lock on background. Acceptance: biometrics gate the stored token (not a bool flag); fallback works; errors handled; runs.

Session Management (Storage, Guards, Logout, Lifecycle)

> Session management ties it all together: securely store tokens, expose **reactive auth state** that drives route guards and UI, refresh transparently, and **fully clear** everything on logout/expiry — behind a single `AuthRepository` the app depends on.

Introduction

The capstone concept: the end-to-end session lifecycle. This file integrates secure storage (15), the refresh interceptor (02_token_auth_jwt.md), route guards (13 · guards_and_redirects), and reactive auth state (11) into one coherent, testable auth backbone.

Why this concept exists

Auth pieces (login, storage, refresh, guards, logout) must work as a *system*. Gaps — tokens not cleared on logout, guards not reacting to state, no expiry handling — cause security and UX bugs. Centralizing the session lifecycle in an `AuthRepository` + reactive state makes it correct and testable.

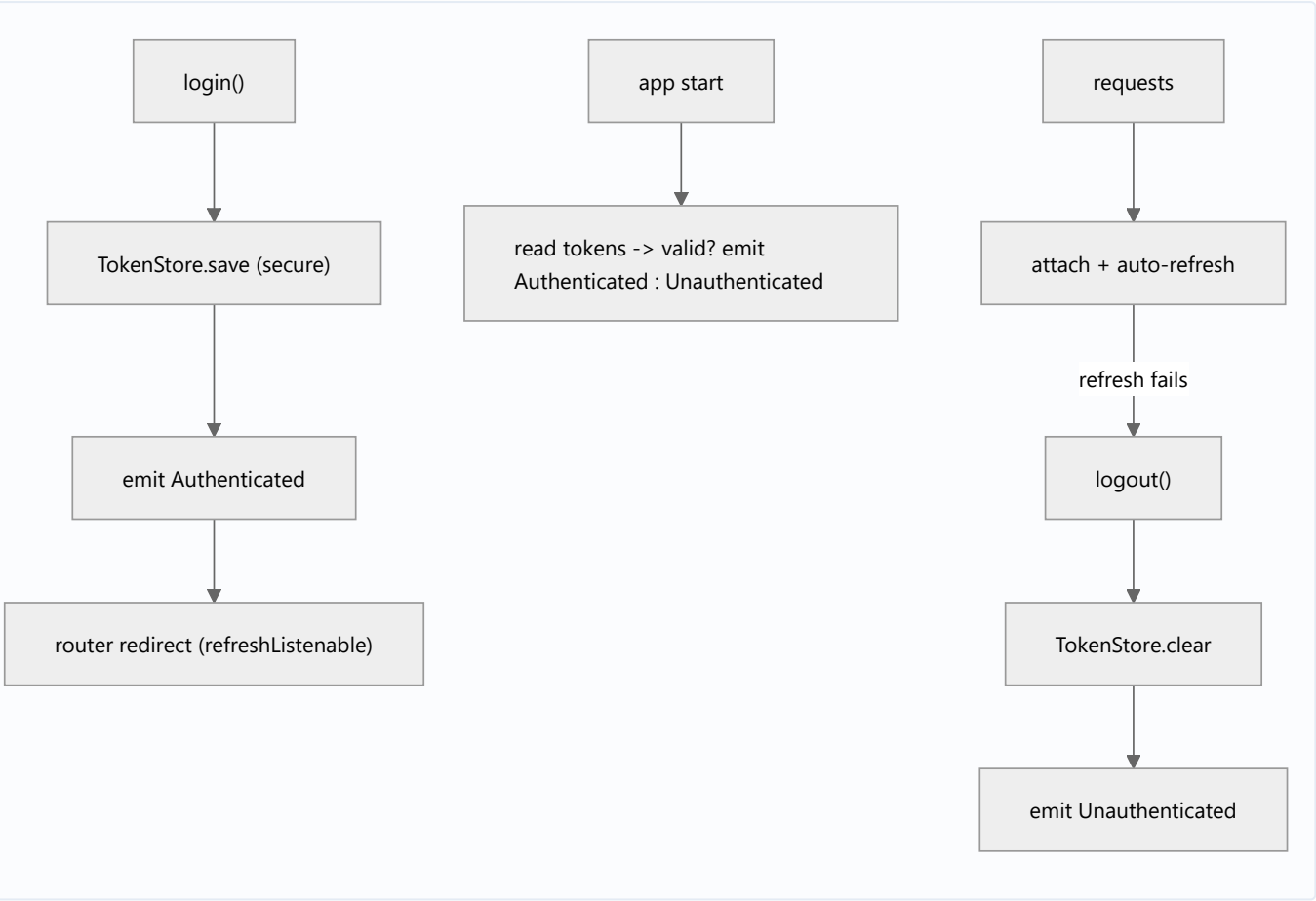
Real-world analogy

A **hotel front desk**: check-in issues a keycard (login→token), the desk tracks your status (auth state), doors read your card (guards), the card auto-renews if you extend (refresh), and checkout deactivates it everywhere (logout clears session). One system, one source of truth.

Problem Statement

Wire: login stores tokens → app exposes `Stream<AuthState>` → guards redirect by state → requests auto-attach/refresh tokens → logout clears tokens + resets state → app restart restores session if valid. You'll assemble the `AuthRepository`-centered lifecycle.

Internal Working



- `AuthRepository` (single source of truth): `login()`, `logout()`, `currentUser`, and a `Stream<AuthState>` / `Listenable` others observe.
- **Storage**: tokens in **secure storage** (15 · secure_storage); on startup, read + validate (expiry/refresh) to restore or clear the session.
- **Reactive state**: expose `AuthState` (sealed: `Authenticated` / `Unauthenticated` / `Loading`); feed it to the router's `refreshListenable` so login/logout re-route instantly (13 · guards_and_redirects).
- **Auto-refresh**: the interceptor handles token expiry transparently; on refresh failure it triggers **logout** (02_token_auth_jwt.md).
- **Logout**: clear **all** tokens/secrets, reset auth state, clear user-scoped caches/DI scopes (14 · scopes_and_lifetimes), and navigate to login.
- **Expiry/idle**: handle server-side revocation (401 that can't refresh) and optional idle timeout; re-lock with biometrics if configured (04_biometric_auth.md).

Memory Representation

Tokens: encrypted in secure storage; briefly in memory when used. Auth state is a small in-memory object broadcast to listeners. Clear user-scoped memory on logout to prevent leaks/data bleed (08 · dispose_and_leaks).

Compiler Behavior

Sealed `AuthState` enables exhaustive UI/guard handling (02 · records_and_patterns).

Runtime Behavior

State changes notify the router (re-run guards) and rebuild auth-dependent UI. Startup restores or clears the session; logout resets everything; refresh failure funnels to logout.

Flutter Engine Behavior / Dart VM Behavior

Not applicable.

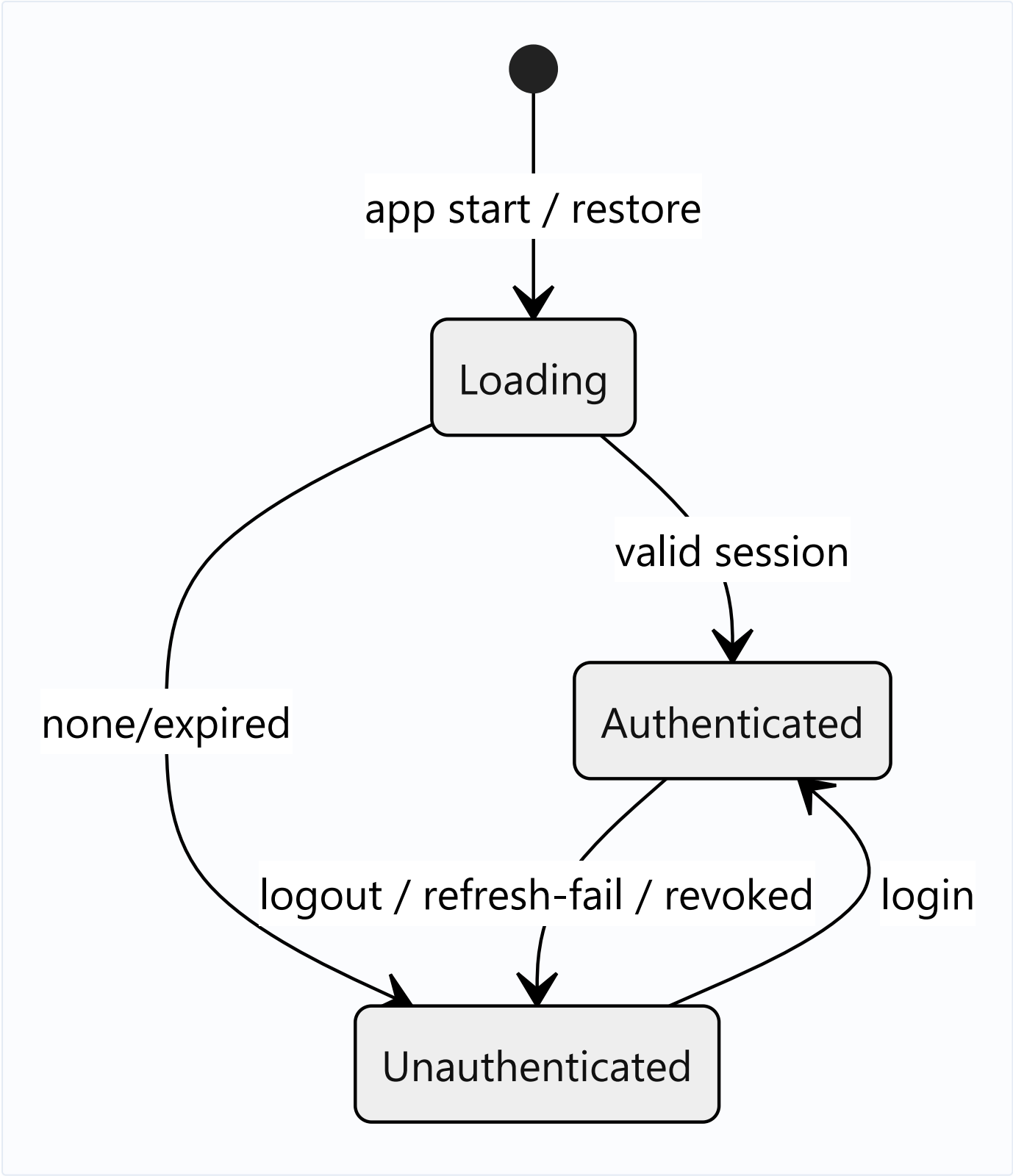
Examples

```
// Reactive auth state + repository (single source of truth)
sealed class AuthState { const AuthState(); }
class AuthLoading extends AuthState { const AuthLoading(); }
class Authenticated extends AuthState { final String userId; const Authenticated(this.userId); }
class Unauthenticated extends AuthState { const Unauthenticated(); }

abstract interface class TokenStore {
  Future<String?> get accessToken;
  Future<void> save({required String access, required String refresh});
  Future<void> clear();
}

class AuthRepository {
  final TokenStore _store;
  final _controller = // broadcast auth state
      // ignore: prefer_const_constructors
      _StateController();
  AuthRepository(this._store);

  Stream<AuthState> get state => _controller.stream;

  Future<void> restore() async {                // app startup
    final token = await _store.accessToken;
    _controller.emit(token != null ? const Authenticated('me') : const Unauthenticated());
    // (validate/refresh expiry in production)
  }
```

```
  Future<void> login(String user, String pass) async {
    _controller.emit(const AuthLoading());
    // ... call API, get tokens ...
    await _store.save(access: 'access', refresh: 'refresh'); // secure storage
    _controller.emit(const Authenticated('me'));
  }

  Future<void> logout() async {
    await _store.clear();                          // clear ALL tokens
    // clear user-scoped caches / DI session scope here
    _controller.emit(const Unauthenticated());
  }
}

// minimal broadcast controller stand-in
class _StateController {
  final _c = // StreamController.broadcast in real code
      <void Function(AuthState)>[];
  final List<AuthState> _last = [const AuthLoading()];
  Stream<AuthState> get stream async* { yield _last.last; }
  void emit(AuthState s) => _last.add(s);
}
```

```
// Router integration (go_router): refreshListenable = auth state,
// redirect: unauthenticated -> /login; authenticated on /login -> /home
// (see 13/guards_and_redirects.md). Logout() emits Unauthenticated -> auto-redirect.
```

Diagrams


app start / restore
Loading
valid session
Authenticated
none/expired
logout / refresh-fail / revoked
login
Unauthenticated

Common Mistakes

Mistake	Why	Fix
Logout that leaves tokens/caches	Security leak, "ghost login"	Clear tokens + user-scoped caches + DI session scope
Guards not reacting to auth changes	Stale routing	Drive guards via `refreshListenable` /state stream
No session restore on startup	Forces re-login every launch	Read/validate tokens at startup
Auth state scattered across app	Inconsistent	Single `AuthRepository` source of truth
Not handling refresh-failure/revocation	Stuck/insecure session	Funnel to logout on unrecoverable 401
User data persisting across accounts	Data bleed	Scope caches to session; clear on logout

Best Practices

- Centralize the lifecycle in an `AuthRepository` with a reactive `AuthState` (single source of truth).
- Store tokens in **secure storage**; **restore/validate** on startup; **auto-refresh** transparently.
- Drive **guards + UI** from auth state (`refreshListenable`); login/logout re-route instantly.
- On logout/expiry/revocation: **clear all tokens, user-scoped caches, and DI session scope**, then route to login.
- Consider **biometric re-lock** and optional **idle timeout** for sensitive apps.
- Keep it **testable**: inject `TokenStore` /API; assert state transitions.

Performance

Session ops are infrequent (login/refresh/logout); reactive state changes are cheap. Clearing caches/scopes on logout prevents memory bleed (Module 21).

Advantages / Disadvantages

- **+** Coherent, secure, reactive session; instant guard/UI updates; clean logout; restore-on-launch; testable.
- **−** Several moving parts to integrate correctly; must be disciplined about clearing everything and handling edge cases (revocation, multi-account).

Interview Questions

1. 🟢 **What does session management encompass?** — Storing tokens, exposing auth state, guarding routes, refreshing, and clearing everything on logout/expiry.

2. 🟢 **Where should the session's source of truth live?** — In a single `AuthRepository` exposing reactive `AuthState`.
3. 🟡 **How do guards react to login/logout instantly?** — The router's `refreshListenable` observes auth state; state changes re-run redirects (13).
4. 🟡 **What must logout do?** — Clear all tokens/secrets, reset auth state, clear user-scoped caches/DI session scope, and route to login.
5. 🟡 **How is a session restored on app launch?** — Read tokens from secure storage, validate/refresh expiry, then emit `Authenticated` / `Unauthenticated`.
6. 🔴 **How do refresh failure and server revocation flow through the system?** — An unrecoverable 401 triggers logout (clear + `Unauthenticated`), which re-routes to login via guards.
7. 🔴 **How do you prevent data bleed between accounts?** — Scope caches/DI to the session and clear them on logout so the next user starts clean.

Senior Engineer Tips

- Make **logout ruthless**: one method that clears tokens, caches, DI session scope, and in-memory state — audited so nothing lingers.
- Drive *everything* auth-related off the single `AuthState` stream; avoid duplicate auth flags scattered around.
- Test the full lifecycle (login→restore→refresh→logout→revocation) with injected fakes; these transitions are where bugs hide.

Architect Perspective

Session management is the integration point of the entire auth stack — storage, tokens/refresh, guards, and reactive state — centralized in an `AuthRepository`. A single-source-of-truth, reactive, ruthlessly-clearing session lifecycle is a security and UX cornerstone that composes with routing, DI scopes, offline, and monitoring (Modules 13, 14, 19, 52).

Summary

- Centralize the session lifecycle in an `AuthRepository` with reactive `AuthState`.
- Secure-store tokens, restore on startup, auto-refresh, and drive guards/UI from state.
- Logout/expiry clears everything (tokens + caches + DI scope) and re-routes to login; keep it testable.

Revision Notes

- `AuthRepository` = single source of truth; `Stream<AuthState>` (sealed) drives guards (`refreshListenable`) + UI.
- Tokens in secure storage; restore/validate on startup; auto-refresh; refresh-fail/revoke → logout.
- Logout clears tokens + user caches + DI session scope + state → route to login.
- Optional biometric re-lock/idle timeout; test the full lifecycle with fakes.

Practice Questions

1. What are all the things logout must clear?
2. How do route guards respond instantly to login/logout?
3. How is a valid session restored on app restart?

Coding Questions

1. Build an `AuthRepository` (login/logout/restore) emitting `AuthState`, backed by a secure `TokenStore`.
2. Wire it to `go_router` guards via `refreshListenable`.
3. Test transitions: login→authenticated, logout→unauthenticated, refresh-fail→logout.

Mini Project — Module capstone

End-to-end session (Flutter): Assemble the full auth backbone: `AuthRepository` (reactive `AuthState`) + secure `TokenStore` + refresh interceptor (02_token_auth_jwt.md) + `go_router` guards (13) + optional biometric unlock (04_biometric_auth.md). Login stores tokens; startup restores; guards react; refresh-failure/logout clears everything (tokens + caches + DI scope) and routes to login. Test the lifecycle with fakes. Acceptance: single source of truth; secure storage; reactive guards; ruthless logout; session restore; tests pass.