**AUTH SERVICE FOR SUPABASE CODE:**

import 'package:supabase\_flutter/supabase\_flutter.dart';

class AuthService {

  final GoTrueClient \_auth = Supabase.instance.client.auth;

  Future<AuthResponse> signIn({

    required String email,

    required String password,

  }) async {

    return await \_auth.signInWithPassword(

      email: email,

      password: password,

    );  }

  Future<AuthResponse> signUp({

    required String email,

    required String password,

  }) async {

    return await \_auth.signUp(

      email: email,

      password: password,

    ); }

  Future<void> signOut() async {

    await \_auth.signOut();

  } }

**CODE BREAKDOWN:**

This AuthService is a small wrapper around Supabase’s auth client (GoTrueClient). It exposes three methods:

* signIn(...) — sign in with email + password
* signUp(...) — register with email + password
* signOut() — sign the current user out

Each method is async and talks to Supabase via \_auth.

**Line-by-line explanation**

import 'package:supabase\_flutter/supabase\_flutter.dart';

Imports Supabase’s Flutter/Dart SDK. That package contains Supabase, GoTrueClient, AuthResponse, and other auth/database helpers.

class AuthService { final GoTrueClient \_auth = Supabase.instance.client.auth; }

* AuthService — a simple service class to centralize auth calls.
* \_auth is a GoTrueClient instance (the Supabase auth client).
* The leading underscore (\_auth) makes the field **private to the Dart file** (Dart’s privacy convention). Other files cannot access \_auth directly.

**signIn method**

Future<AuthResponse> signIn({

required String email,

required String password,

}) async {

return await \_auth.signInWithPassword(

email: email,

password: password,

);

}

* **Signature:** returns a Future<AuthResponse> — an async result object from Supabase.
* required String email, required String password — these are named parameters and must be provided (null-safety + clarity).
* \_auth.signInWithPassword(...) — calls Supabase to authenticate.
* AuthResponse typically contains fields like session, user, and error.
  + On success response.session (and response.user) will be set.
  + On failure response.error will be set with details.

**signUp method**

Future<AuthResponse> signUp({

required String email,

required String password,

}) async {

return await \_auth.signUp(

email: email,

password: password,

);

}

* Calls Supabase to create a new user.
* Depending on your Supabase project settings, sign up may:
  + create a user and return a session immediately, or
  + require email confirmation (in which case session might be null until email is confirmed).
* Check response.error to see if sign-up failed (weak password, duplicate email, etc).

**signOut method**

Future<void> signOut() async {

await \_auth.signOut();

}

* Signs out the current session.
* Supabase SDK will clear persisted session/token for you (SDK handles persistence and refresh tokens under the hood).

**How you’d use it in Flutter (example)**

final authService = AuthService();

// Sign in

final res = await authService.signIn(email: 'me@ex.com', password: 'pass123');

if (res.error != null) {

// show error: res.error!.message

} else {

// success: res.session / res.user available

Navigator.pushReplacement(...);

}

For sign up, same approach: call signUp(...), inspect res.error, and if no error check res.user or res.session. If your project requires email confirm, show a message asking the user to confirm email.

**Good practices / improvements you should consider**

1. **Error handling:** wrap calls in try/catch or inspect response.error and convert to user-friendly messages.
2. **Return a typed result:** instead of returning raw AuthResponse, consider a Result wrapper or throw custom exceptions so your UI doesn’t need to know SDK internals.
3. **Loading states:** perform signIn/signUp inside a loading state in your UI (avoid duplicate taps).
4. **Listen for auth changes:** use Supabase.instance.client.auth.onAuthStateChange (or equivalent) to react when session changes (e.g., token refresh, sign-out from another device).
5. **Security:** don’t put **service\_role** keys on the client. The client (mobile/web) may safely have the **anon** key; secret keys must stay on servers.
6. **Email confirmation flows:** handle the “confirm your email” state in UI since signUp might not create an active session depending on settings.
7. **Dependency injection / singleton:** provide AuthService via Provider/Riverpod/GetIt so you can test and mock it easily.
8. **Unit tests:** mock GoTrueClient or wrap it behind an interface to test logic.

**Example: improved method with basic error handling**

Future<void> signInOrThrow({

required String email,

required String password,

}) async {

final res = await \_auth.signInWithPassword(email: email, password: password);

if (res.error != null) {

throw Exception(res.error!.message); // or a custom AuthException

}

if (res.session == null) {

throw Exception('No session returned after sign in.');

}

}

This pattern lets your UI just try { await authService.signInOrThrow(...) } catch (e) { show error }.