**LOGIN SCREEN OF ALL\_NEWS\_INFORMATION:CODE:**

import 'package:flutter/material.dart';

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

import 'auth\_service.dart';

import 'home\_screen.dart';

import 'signup\_screen.dart';

import 'profile\_service.dart';

class LoginScreen extends StatefulWidget {

  const LoginScreen({super.key});

  @override

  State<LoginScreen> createState() => \_LoginScreenState();

}

class \_LoginScreenState extends State<LoginScreen> {

  final \_formKey = GlobalKey<FormState>();

  final \_emailController = TextEditingController();

  final \_passwordController = TextEditingController();

  final \_authService = AuthService();

  final \_profileService = ProfileService();

  bool \_isLoading = false; // ADD LOADING STATE

  Future<void> \_login() async {

    if (\_formKey.currentState!.validate()) {

      setState(() => \_isLoading = true); // START LOADING

      try {

        final response = await \_authService.signIn(

          email: \_emailController.text,

          password: \_passwordController.text,

        );

        if (response.user != null) {

          // VERIFY USER EXISTS IN PUBLIC.USERS TABLE

          await \_verifyUserInPublicTables(response.user!);

          Navigator.of(context).pushReplacement(

            MaterialPageRoute(builder: (context) => const HomeScreen()),

          );

        }

      } on AuthException catch (e) {

        ScaffoldMessenger.of(

          context,

        ).showSnackBar(SnackBar(content: Text(e.message)));

      } catch (e) {

        // ADD GENERAL ERROR CATCH

        ScaffoldMessenger.of(

          context,

        ).showSnackBar(SnackBar(content: Text('Login error: $e')));

      } finally {

        // STOP LOADING

        setState(() => \_isLoading = false);

      }

    }

  }

  // ADD VERIFICATION METHOD

  Future<void> \_verifyUserInPublicTables(User user) async {

    try {

      final profile = await \_profileService.getCompleteUserProfile(user.id);

      if (profile != null) {

        print('✅ Login successful - User found in public tables: $profile');

      } else {

        print('⚠️ User not found in public tables, but auth login successful');

      }

    } catch (e) {

      print('⚠️ Error checking public tables: $e');

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: const Text('Login')),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Form(

          key: \_formKey,

          child: Column(

            mainAxisAlignment: MainAxisAlignment.center,

            children: [

              TextFormField(

                controller: \_emailController,

                decoration: const InputDecoration(labelText: 'Email'),

                validator: (value) {

                  if (value == null || value.isEmpty) {

                    return 'Please enter your email';

                  }

                  return null;

                },

              ),

              TextFormField(

                controller: \_passwordController,

                decoration: const InputDecoration(labelText: 'Password'),

                obscureText: true,

                validator: (value) {

                  if (value == null || value.isEmpty) {

                    return 'Please enter your password';

                  }

                  return null;

                },

              ),

              const SizedBox(height: 20),

              ElevatedButton(

                onPressed: \_isLoading ? null : \_login,

                child: \_isLoading

                    ? const SizedBox(

                        height: 20,

                        width: 20,

                        child: CircularProgressIndicator(strokeWidth: 2),

                      )

                    : const Text('Login'),

              ),

              TextButton(

                onPressed: () {

                  Navigator.of(context).push(

                    MaterialPageRoute(

                      builder: (context) => const SignupScreen(),

                    ),

                  );

                },

                child: const Text('Don\'t have an account? Sign up'),

              ),

            ],

          ),

        ),

      ),

    );

  }

}

**BREAKDOWN**

**1) What the imports & fields do**

import 'package:flutter/material.dart';

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

import 'auth\_service.dart';

import 'home\_screen.dart';

import 'signup\_screen.dart';

import 'profile\_service.dart';

* flutter/material.dart: Flutter UI toolkit.
* supabase\_flutter.dart: Supabase client types (including AuthException used later).
* auth\_service.dart and profile\_service.dart: your wrapper classes that talk to Supabase — they encapsulate sign-in, sign-up, profile creation/updates, etc. (not shown here).
* home\_screen.dart, signup\_screen.dart: screens to navigate to after a successful login or to switch to sign-up.

Fields inside \_LoginScreenState:

final \_formKey = GlobalKey<FormState>();

final \_emailController = TextEditingController();

final \_passwordController = TextEditingController();

final \_authService = AuthService();

final \_profileService = ProfileService();

bool \_isLoading = false;

* \_formKey: used to validate the Form (so you can call \_formKey.currentState!.validate() and run your TextFormField validators).
* \_emailController, \_passwordController: read/write the input typed by user.
* \_authService: an instance of your authentication helper (likely wraps Supabase.instance.client.auth.signIn() etc.).
* \_profileService: service used to read/write profile rows in your public.profiles or public.users table.
* \_isLoading: UI state flag to show a loading spinner / disable button while the sign-in request is in progress.

**2) The \_login() method — step-by-step**

Future<void> \_login() async {

if (\_formKey.currentState!.validate()) {

setState(() => \_isLoading = true); // START LOADING

try {

final response = await \_authService.signIn(

email: \_emailController.text,

password: \_passwordController.text,

);

if (response.user != null) {

// VERIFY USER EXISTS IN PUBLIC.USERS TABLE

await \_verifyUserInPublicTables(response.user!);

Navigator.of(context).pushReplacement(

MaterialPageRoute(builder: (context) => const HomeScreen()),

);

}

} on AuthException catch (e) {

ScaffoldMessenger.of(

context,

).showSnackBar(SnackBar(content: Text(e.message)));

} catch (e) {

ScaffoldMessenger.of(

context,

).showSnackBar(SnackBar(content: Text('Login error: $e')));

} finally {

setState(() => \_isLoading = false); // STOP LOADING

}

}

}

* if (\_formKey.currentState!.validate())  
  Runs validators attached to the Form fields. Only proceeds if validation passes (e.g., email format, password length).
* setState(() => \_isLoading = true);  
  Start loading state — used to show spinner and prevent double submissions.
* final response = await \_authService.signIn(...)  
  Calls your auth wrapper. Typically this does something like supabase.auth.signInWithPassword(email, password) and returns an auth response object that contains user and session info.
* if (response.user != null)  
  Checks that login succeeded and a User object exists. If null, login failed silently — you should surface an error in that case (see suggestions).
* await \_verifyUserInPublicTables(response.user!);  
  This function (not shown) should ensure the user has a row in your public.profiles (or public.users) table — e.g., create a profile row if missing. This ensures app-level profile data exists for that auth.uid().
* Navigator.of(context).pushReplacement(...)  
  Navigate to HomeScreen, replacing the current screen in the stack (user cannot go back to login with the back button).
* on AuthException catch (e)  
  Catches Supabase-specific auth error (invalid credentials, no user, etc.) and shows e.message in a SnackBar.
* catch (e)  
  Generic catch for any other exception; shows a SnackBar with Login error: $e.
* finally { setState(() => \_isLoading = false); }  
  Ensures loading state is turned off regardless of success or error.

**3) Important practical notes & potential issues**

* **Dispose controllers**: you must override dispose() and call \_emailController.dispose(); \_passwordController.dispose(); to avoid memory leaks.
* **Check mounted before calling Navigator / setState**: if the widget is unmounted during the async call, calling setState or Navigator will throw. Wrap with if (!mounted) return; before setState or navigation.
* **Handle the response.user == null case**: currently nothing appears if response.user is null — add an error SnackBar there.
* **Disable the login button while \_isLoading is true**: prevents duplicate submits.
* **Show clear feedback**: show a CircularProgressIndicator in the button or overlay so users know something is happening.
* **Trim user input**: use \_emailController.text.trim() when passing to sign-in to avoid leading/trailing spaces.
* **AuthException**: fine to catch separately — it gives user-friendly messages from Supabase.

**4) Suggested small improvements (code)**

Add dispose, mounted checks, and better error handling. Drop this directly into your class.

@override

void dispose() {

\_emailController.dispose();

\_passwordController.dispose();

super.dispose();

}

Future<void> \_login() async {

if (!\_formKey.currentState!.validate()) return;

setState(() => \_isLoading = true);

try {

final response = await \_authService.signIn(

email: \_emailController.text.trim(),

password: \_passwordController.text,

);

if (response.user == null) {

if (!mounted) return;

ScaffoldMessenger.of(context).showSnackBar(

const SnackBar(content: Text('Login failed: user not returned')),

);

return;

}

await \_verifyUserInPublicTables(response.user!);

if (!mounted) return;

Navigator.of(context).pushReplacement(

MaterialPageRoute(builder: (context) => const HomeScreen()),

);

} on AuthException catch (e) {

if (!mounted) return;

ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text(e.message)));

} catch (e) {

if (!mounted) return;

ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Login error: $e')));

} finally {

if (!mounted) return;

setState(() => \_isLoading = false);

}

}

And in your login button widget:

ElevatedButton(

onPressed: \_isLoading ? null : \_login,

child: \_isLoading

? const SizedBox(width:24, height:24, child: CircularProgressIndicator(strokeWidth:2))

: const Text('Login'),

),

**5) UX tips for login flows**

* After successful login, you may want to preload profile data (call ProfileService.fetchProfile()), and pass it into HomeScreen so HomeScreen can render immediately.
* Use secure text for password fields (obscureText: true).
* Consider adding FocusScope.of(context).unfocus() before starting login so keyboard dismisses.
* On auto-login / session restore, call the same \_verifyUserInPublicTables() once at app start to ensure profile exists.

**6) Quick summary**

* Your \_login() is well-structured: validation → loading → auth call → profile check → navigation → proper error handling → finally stop loading.
* Add dispose(), mounted checks, handle response.user == null, and disable UI while loading to harden it.

this is **Part 2** of your login\_screen.dart, and it’s basically a **post-login verification function** to ensure the authenticated user’s data also exists in your **public tables** (like public.profiles or public.users in Supabase).

Let’s go through it **step by step in detail**, including what happens behind the scenes 👇

**🧩 The Full Code**

// ADD VERIFICATION METHOD

Future<void> \_verifyUserInPublicTables(User user) async {

try {

final profile = await \_profileService.getCompleteUserProfile(user.id);

if (profile != null) {

print('✅ Login successful - User found in public tables: $profile');

} else {

print('⚠️ User not found in public tables, but auth login successful');

}

} catch (e) {

print('⚠️ Error checking public tables: $e');

}

}

**🔍 Line-by-Line Explanation**

**1️⃣ Future<void> \_verifyUserInPublicTables(User user) async {**

* This defines an **asynchronous private method** (that returns nothing — void) named \_verifyUserInPublicTables.
* It takes one argument — a User object from Supabase (the user who just logged in).
* The async keyword allows using the await keyword inside (for asynchronous calls to Supabase).

👉 Purpose:  
After the user logs in, this function confirms whether the same user exists in your **custom user table**, not just in Supabase’s internal authentication system.

**2️⃣ try {**

Starts a try-catch block to safely handle errors that may occur (like network failure or invalid query).

**3️⃣ final profile = await \_profileService.getCompleteUserProfile(user.id);**

* \_profileService is your custom service class (from profile\_service.dart) that manages user profiles in the database.
* getCompleteUserProfile(user.id) probably runs a Supabase query like:
* final response = await Supabase.instance.client
* .from('profiles')
* .select()
* .eq('id', userId)
* .single();

It returns the user’s profile record if found.

✅ If a record is found → returns profile data (like name, email, phone).  
❌ If not found → returns null.

Perfect bro 🔥 — now let’s break down the **Part 3 (the UI section)** of your login\_screen.dart file — this is where the **actual login form** (with fields, buttons, and navigation) is created and displayed on screen.

You’ve already covered the backend logic (login + verification), so now this is how the UI connects to that logic.

**🧠 Detailed Explanation (Step-by-Step)**

**1️⃣ @override**

* This tells Flutter that we are **overriding** the build() method from the State class (\_LoginScreenState).
* It defines **what will appear on the screen**.

**2️⃣ Scaffold(...)**

* The **base structure** of the screen.
* Provides the layout skeleton — app bar, body, floating action buttons, etc.
* Here, it creates a screen with:
* appBar: AppBar(title: const Text('Login')),

→ This gives you a top app bar titled “Login”.

**3️⃣ body: Padding(...)**

* Adds spacing **around** the form so the elements don’t touch the screen edges.
* padding: const EdgeInsets.all(16.0)

means 16 pixels of padding on all sides.

**4️⃣ Form(...)**

* The entire login form is wrapped inside a **Form widget**.
* It is connected to \_formKey, so you can **validate all fields together** when the user presses the login button.

**5️⃣ Column(...)**

* Aligns widgets vertically (like stacking them one under the other).
* mainAxisAlignment: MainAxisAlignment.center centers the form content vertically on the screen.

**6️⃣ Email Field**

TextFormField(

controller: \_emailController,

decoration: const InputDecoration(labelText: 'Email'),

validator: (value) {

if (value == null || value.isEmpty) {

return 'Please enter your email';

}

return null;

},

),

🔹 **controller** — \_emailController stores and retrieves what user types in the email field.  
🔹 **decoration** — adds label “Email”.  
🔹 **validator** — checks if input is empty. If yes, it returns an error message (used when \_formKey.currentState!.validate() runs).

**7️⃣ Password Field**

TextFormField(

controller: \_passwordController,

decoration: const InputDecoration(labelText: 'Password'),

obscureText: true,

validator: (value) {

if (value == null || value.isEmpty) {

return 'Please enter your password';

}

return null;

},

),

🔹 Uses \_passwordController for managing input.  
🔹 obscureText: true hides the password with bullets (••••).  
🔹 Validator ensures the field isn’t empty.

**8️⃣ const SizedBox(height: 20)**

Adds vertical space (20px) between the input fields and button — for better UI spacing.

**9️⃣ Login Button**

ElevatedButton(

onPressed: \_isLoading ? null : \_login,

child: \_isLoading

? const SizedBox(

height: 20,

width: 20,

child: CircularProgressIndicator(strokeWidth: 2),

)

: const Text('Login'),

),

🔹 **onPressed**

* If \_isLoading is true, the button is disabled (null).
* Otherwise, pressing it calls the \_login() function (we explained that in Part 1).

🔹 **child**

* If \_isLoading is true → shows a small circular loading spinner inside the button.
* Else → shows simple text “Login”.

✅ This is how your app visually indicates to the user that login is in progress (prevents multiple taps).

**🔟 Signup Navigation Button**

TextButton(

onPressed: () {

Navigator.of(context).push(

MaterialPageRoute(

builder: (context) => const SignupScreen(),

),

);

},

child: const Text('Don\'t have an account? Sign up'),

),

* This creates a **text button** below the login button.
* When clicked, it navigates to the **SignupScreen** (your registration page).
* It uses Navigator.of(context).push() to open a new route (screen).

**🎯 Final Output Summary**

When you run this screen:

1. You see:
   * App bar with **Login** title.
   * Two input fields → Email and Password.
   * A Login button (shows a loading spinner when pressed).
   * A small text button linking to the Signup page.
2. When user presses **Login**:
   * \_login() runs → validates form → calls Supabase sign-in.
   * If successful → \_verifyUserInPublicTables() checks database.
   * Then navigates to **HomeScreen**.

**Question: Bro At "class LoginScreen extends StatefulWidget { const LoginScreen({super.key}); @override State<LoginScreen> createState() => \_LoginScreenState(); }" What is 'State<LoginScreen>'? Why we use 'const LoginScreen({super.key});' command? Why we extend 'StatefulWidget' in LoginScreen? Why we use '@override' above 'State<LoginScreen>'?**