CP Repository

Notes on the Project: Explanation of Code & Database.

(For a better reading experience, use a PC or tablet instead of a mobile screen.)

03 FEB 2026

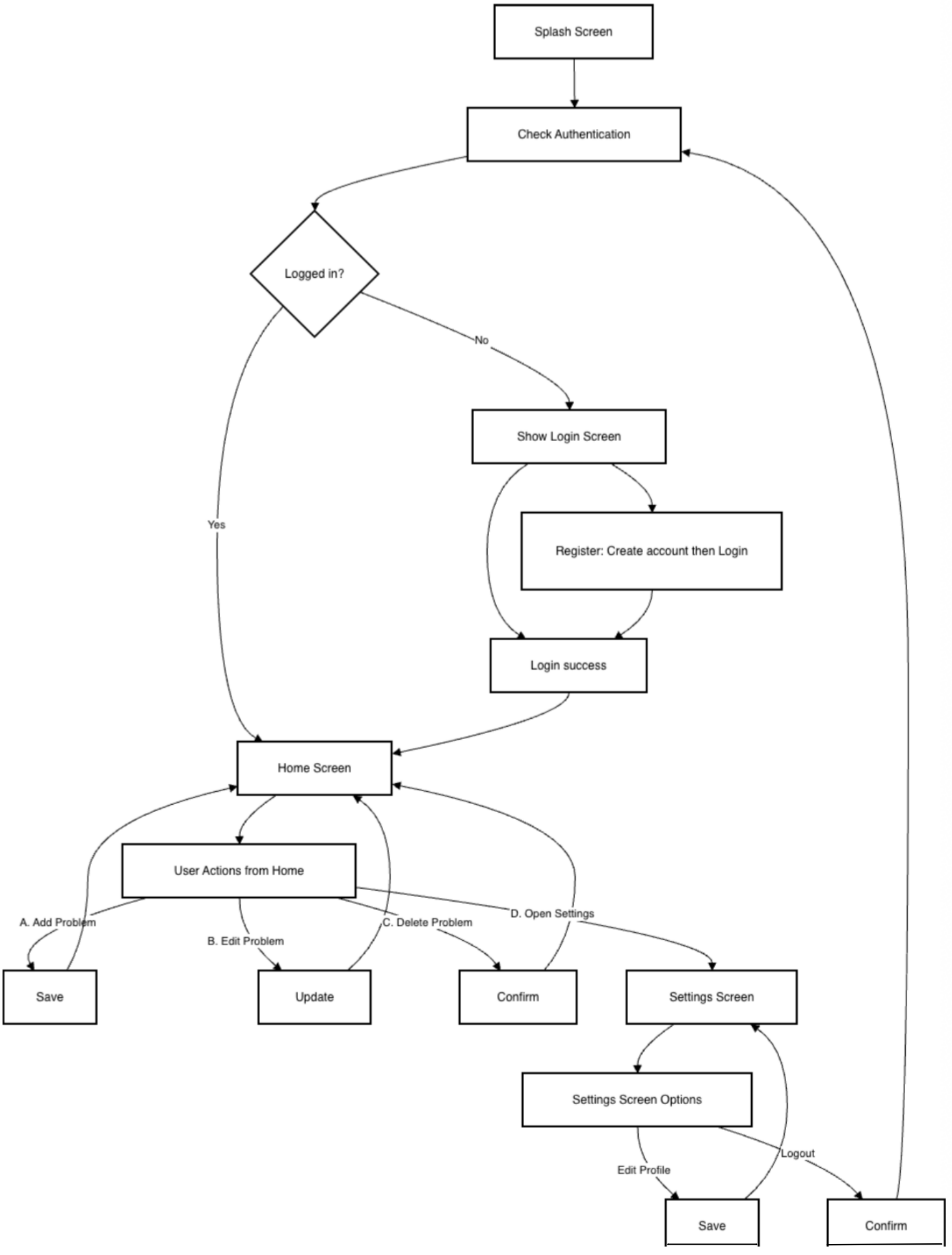
Project Overview

```
CP Problem Tracker
├── User Authentication (Email/Password)
├── Problems Database (CRUD operations)
├── User Profiles (Name/University)
├── Statistics Dashboard
└── Focus Feature (Single pinned problem)
```

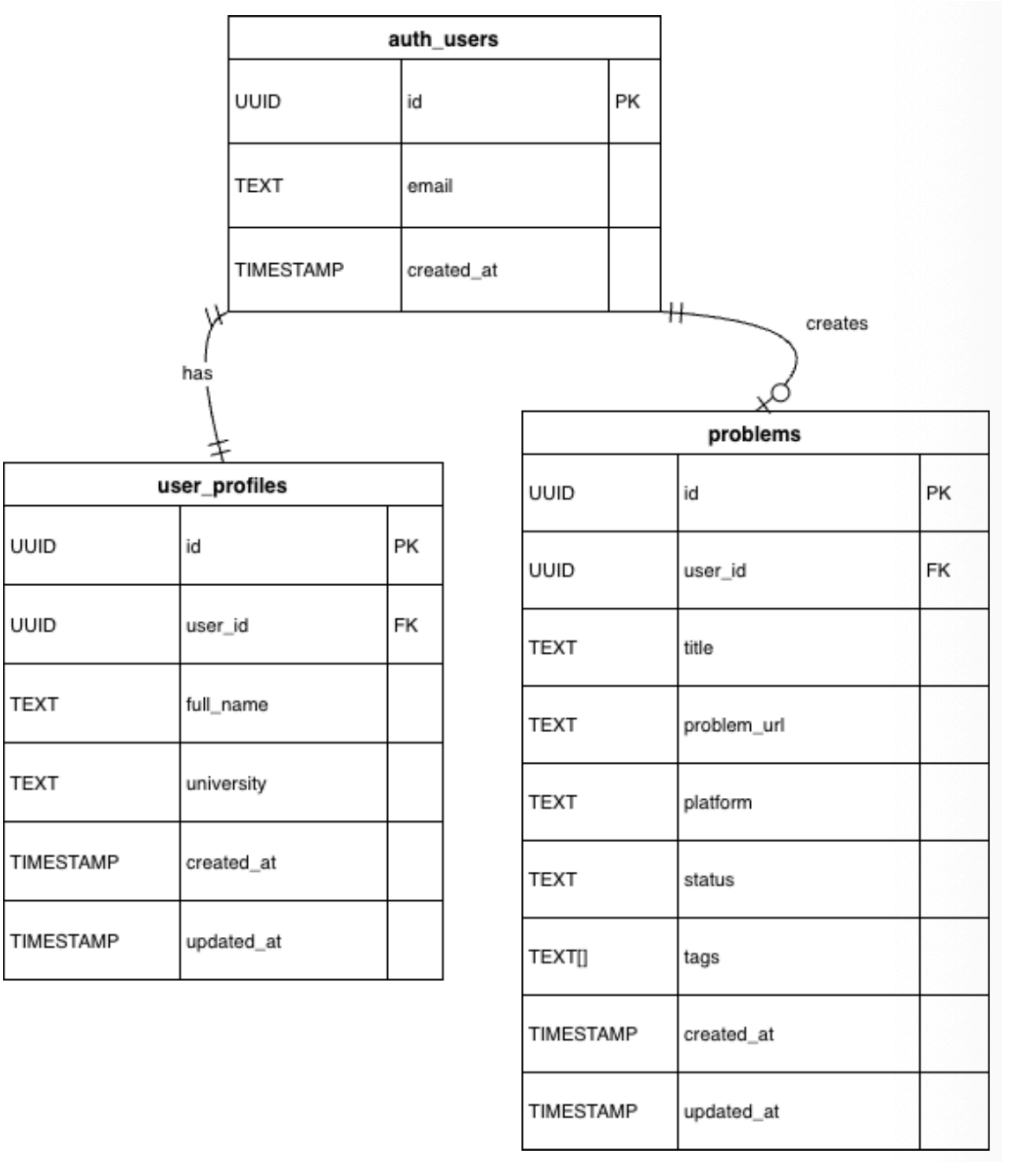
Technology Stack

```
Frontend: Flutter (Dart)
Backend: Supabase (PostgreSQL + Auth)
Database: 2 Tables + RLS Policies
Deployment: Supabase Cloud
```

Workflow Diagram



ER Diagram


auth_users
UUID
id
PK
TEXT
email
TIMESTAMP
created_at
has
creates
user_profiles
UUID
id
PK
UUID
user_id
FK
TEXT
full_name
TEXT
university
TIMESTAMP
created_at
TIMESTAMP
updated_at
problems
UUID
id
PK
UUID
user_id
FK
TEXT
title
TEXT
problem_url
TEXT
platform
TEXT
status
TEXT[]
tags
TIMESTAMP
created_at
TIMESTAMP
updated_at

ER Diagram(cleanner)

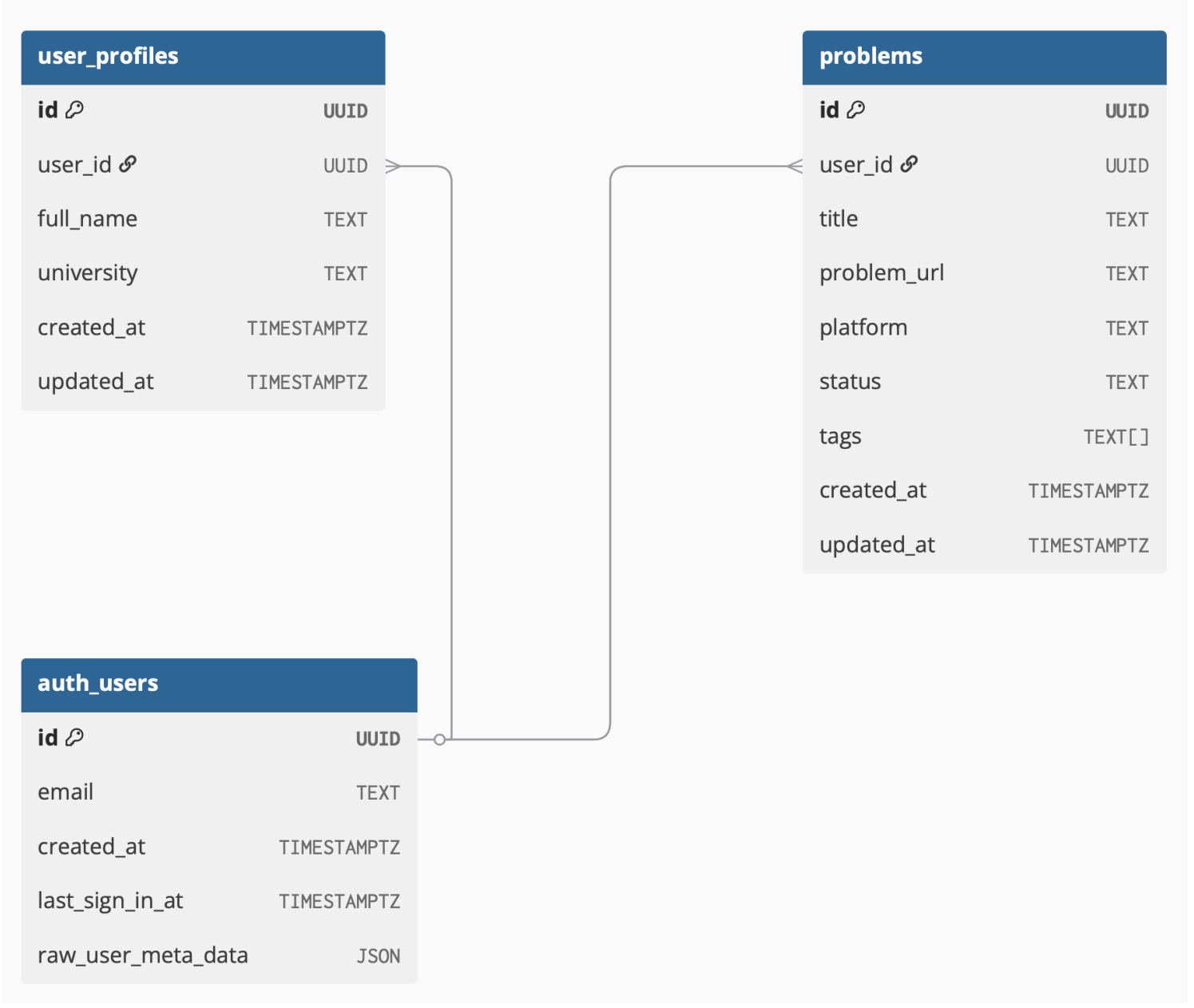
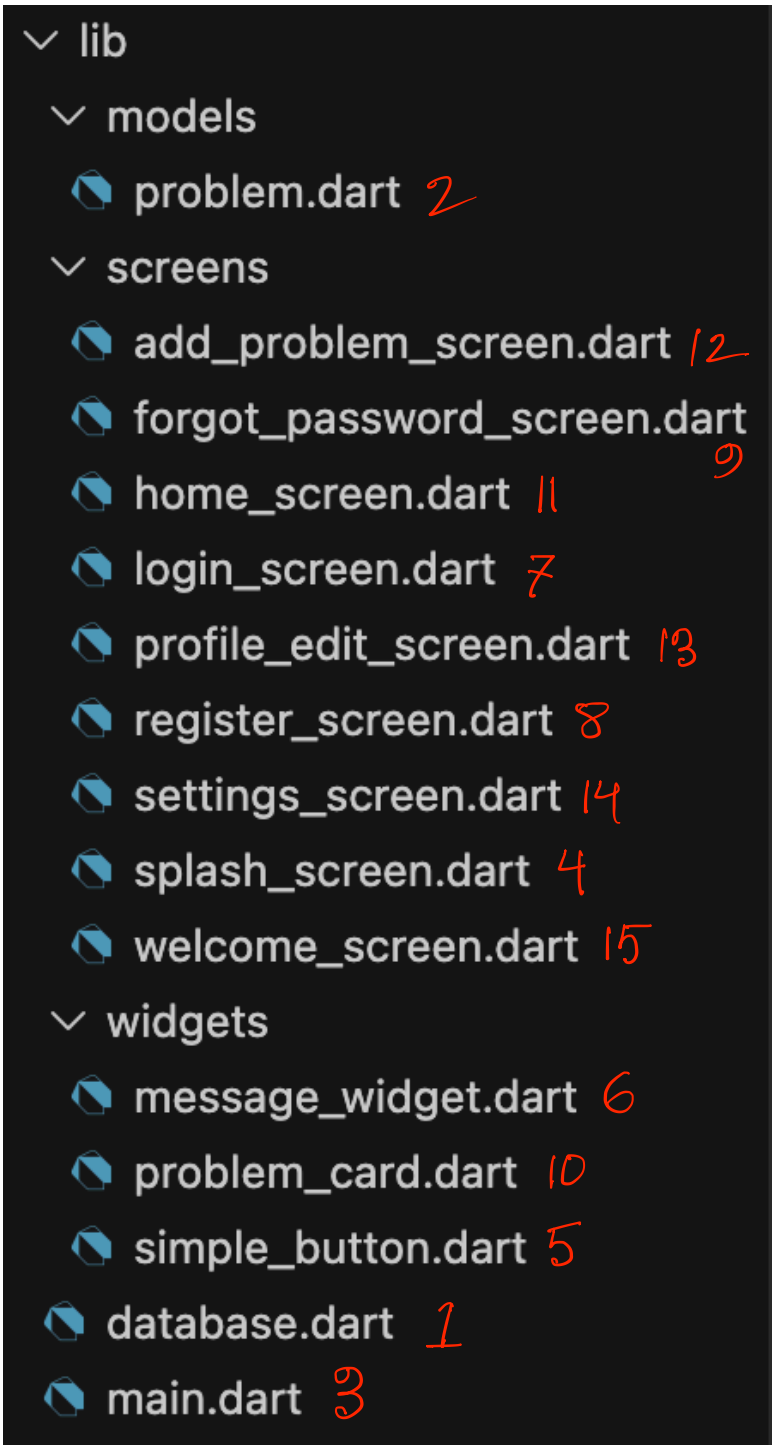


Table of Contents



database.dart

- final == const

```dart
import 'package:supabase_flutter/supabase_flutter.dart';
import 'package:url_launcher/url_launcher.dart';
import 'models/problem.dart';

final supabase = Supabase.instance.client;

String? getUserId() {
  return supabase.auth.currentUser?.id;
}

bool isLoggedIn() {
  return supabase.auth.currentUser != null;
}

Future<String?> register(String email, String password) async {
  try {
    final response = await supabase.auth.signUp(
      email: email,
      password: password,
    );
    return response.user?.id;
  } catch (e) {
    throw e.toString();
  }
}

Future<String?> login(String email, String password) async {
  try {
    final response = await supabase.auth.signInWithPassword(
      email: email,
      password: password,
    );
    return response.user?.id;
  } catch (e) {
    throw e.toString();
  }
}

Future<void> logout() async {
  await supabase.auth.signOut();
}
```

opening links

gateway to the entire backend

users UUID or null

Splash screen decides where to navigate

a value will be available later. Asynchronous.

error handling

creates user in auth.users table

user id

similar to register

Clears JWT token

User becomes unauth.

JWT → JSON Web Token

RLS → Row-Level Security

req. to access table fail for RLS Policy

```dart
Future<List<Problem>> getProblems() async {
  final userId = getUserId();
  if (userId == null) return [];

  try {
    final response = await supabase
        .from('problems')
        .select()
        .eq('user_id', userId)
        .order('created_at', ascending: false);

    List<Problem> problems = [];
    for (var item in response) {
      problems.add(Problem.fromMap(item));
    }
    return problems;
  } catch (e) {
    return [];
  }
}

Future<void> addProblem(Problem problem) async {
  final userId = getUserId();
  if (userId == null) throw 'Not logged in';

  final data = problem.toMap();
  data['user_id'] = userId;
  await supabase.from('problems').insert(data);
}

Future<void> updateProblemStatus(String problemId, String newStatus) async {
  final updateData = {'status': newStatus};

  await supabase
      .from('problems')
      .update(updateData)
      .eq('id', problemId);
}

Future<void> updateProblemTags(String problemId, List<String> tags) async {
  await supabase
      .from('problems')
      .update({'tags': tags})
      .eq('id', problemId);
}
```

easy

table name
WHERE user_id =: userID
ORDER BY created_at DESC

```dart
static Problem fromMap(Map<String, dynamic> data) {
```

#fromMap একটা method, map input নেয়।
এবং Problem return করে।

#তার মানে 'item' হলো একেকটা map.

```dart
return Problem(
  id: data['id']?.toString() ?? '',
  userId: data['user_id']?.toString() ?? '',
  title: data['title']?.toString() ?? '',
  url: data['problem_url']?.toString() ?? '',
  platform: data['platform']?.toString() ?? '',
  tags: tagsList,
  status: statusText,
  createdAt: data['created_at'] != null
      ? DateTime.parse(data['created_at'].toString())
      : DateTime.now(),
);
```

#user এর info map এ convert

c++ equivalent of data

```dart
std::map<std::string,std::string> data;
data["title"]="DP Problem";
data["problem_url"]="https://codeforces
data["platform"]="Codeforces";
data["status"]="Pending";
data["user_id"]="abc123";
```

used in AddProblemScreen.save()

SET status =: newStatus
WHERE id =: problemID

SET tags =: newStatus
WHERE id =: problemID

```dart
Future<void> deleteProblem(String problemId) async {
  await supabase
      .from('problems')
      .delete()
      .eq('id', problemId);
}

Future<Map<String, int>> getStats() async {
  final problems = await getProblems();

  int pending = 0;
  int attempt = 0;
  int solved = 0;

  for (var problem in problems) {
    if (problem.status == 'Pending') pending++;
    else if (problem.status == 'Attempt') attempt++;
    else if (problem.status == 'Solved') solved++;
    else pending++;
  }

  return {
    'total': problems.length,
    'pending': pending,
    'attempt': attempt,
    'solved': solved,
  };
}

Future<void> openUrl(String url) async {
  try {
    final uri = Uri.parse(url);
    if (await canLaunchUrl(uri)) {
      await launchUrl(uri);
    }
  } catch (_) {}
}
```

DELETE FROM problems WHERE id = : problemID

array

easy

```cpp
std::map<std::string,int> getStats(const std::vector<Problem>& problems){
    int pending=0, attempt=0, solved=0;
    for(const auto& problem: problems){
        if(problem.status=="Pending") pending++;
        else if(problem.status=="Attempt") attempt++;
        else if(problem.status=="Solved") solved++;
        else pending++;
    }
    return {
        {"total", (int)problems.size()},
        {"pending", pending},
        {"attempt", attempt},
        {"solved", solved}
    };
}
```

C++ CODE

ইন্টারনালি এটা এরকে text হিসেবে করে।

browser installed? check

যদি url "xyz" হয়, false দিবে।

```dart
Future<Map<String, dynamic>?> getUserProfile() async {
  final userId = getUserId();
  if (userId == null) return null;

  try {
    final response = await supabase
        .from('user_profiles')
        .select()
        .eq('user_id', userId)
        .maybeSingle();

    return response;
  } catch (e) {
    return null;
  }
}

Future<void> updateUserProfile({
  String? fullName,
  String? university,
}) async {
  final userId = getUserId();
  if (userId == null) throw 'Not logged in';

  final data = <String, dynamic>{};
  if (fullName != null) data['full_name'] = fullName.trim();
  if (university != null) data['university'] = university.trim();
  data['updated_at'] = DateTime.now().toIso8601String();

  try {
    final existing = await supabase
        .from('user_profiles')
        .select()
        .eq('user_id', userId)
        .maybeSingle();

    if (existing == null) {
      data['user_id'] = userId;
      await supabase.from('user_profiles').insert(data);
    } else {
      await supabase
          .from('user_profiles')
          .update(data)
          .eq('user_id', userId);
    }
  } catch (e) {
    throw 'Failed to update profile: $e';
  }
}
```

used in SettingScreen

```sql
SELECT *
FROM user_profiles
WHERE user_id = :userId
LIMIT 1;
```

used in ProfileEditScreen.saveProfile()

```sql
SELECT *
FROM user_profiles
WHERE user_id = :userId
LIMIT 1;
```

```sql
INSERT INTO user_profiles (
  user_id,
  full_name,
  university,
  updated_at
)
VALUES (
  :userId,
  :fullName,
  :university,
  :updatedAt
);
```

```sql
UPDATE user_profiles
SET
  full_name = :fullName,
  university = :university,
  updated_at = :updatedAt
WHERE user_id = :userId;
```

problem.dart

```dart
class Problem {
  String id;
  String userId;
  String title;
  String url;
  String platform;
  List<String> tags;
  String status;
  DateTime createdAt;

  Problem({
    this.id = '',
    this.userId = '',
    required this.title,
    required this.url,
    required this.platform,
    this.tags = const [],
    this.status = 'Pending',
    required this.createdAt,
  });

  bool get isValidUrl {
    try {
      final uri = Uri.parse(url);
      return uri.isAbsolute;
    } catch (_) {
      return false;
    }
  }

  bool get isSolved => status.toLowerCase() == 'solved';
```

class defination

Primary key

id = UUID = PK

userID = FK

constructor

DB will generate

added later → curr. logged userID

```
user_id UUID REFERENCES auth.users(id)
```

input

current time

Problem card to make URL clickable

check (https://)

getter

```dart
bool get isSolved {
  String currentStatus = status.toLowerCase();

  if (currentStatus == 'solved') {
    return true;
  } else {
    return false;
  }
}
```

to normal

Arrow function, fat arrow

In Dart, official name

Arrow syntax uses =>

```dart
  static Problem fromMap(Map<String, dynamic> data) {
    String statusText = data['status']?.toString() ?? 'Pending';
    if (statusText.toLowerCase() == 'pending') statusText = 'Pending';
    if (statusText.toLowerCase() == 'attempt') statusText = 'Attempt';
    if (statusText.toLowerCase() == 'solved') statusText = 'Solved';

    List<String> tagsList = [];
    if (data['tags'] != null) {
      if (data['tags'] is List) {
        tagsList = List<String>.from(data['tags']);
      }
    }

    return Problem(
      id: data['id']?.toString() ?? '',
      userId: data['user_id']?.toString() ?? '',
      title: data['title']?.toString() ?? '',
      url: data['problem_url']?.toString() ?? '',
      platform: data['platform']?.toString() ?? '',
      tags: tagsList,
      status: statusText,
      createdAt: data['created_at'] != null
          ? DateTime.parse(data['created_at'].toString())
          : DateTime.now(),
    ); // Problem
  }

  Map<String, dynamic> toMap() {
    final map = <String, dynamic>{
      'title': title.trim(),
      'problem_url': url.trim(),
      'platform': platform.trim(),
      'status': status,
    };

    if (tags.isNotEmpty) {
      map['tags'] = tags;
    }

    return map;
  }
}
```

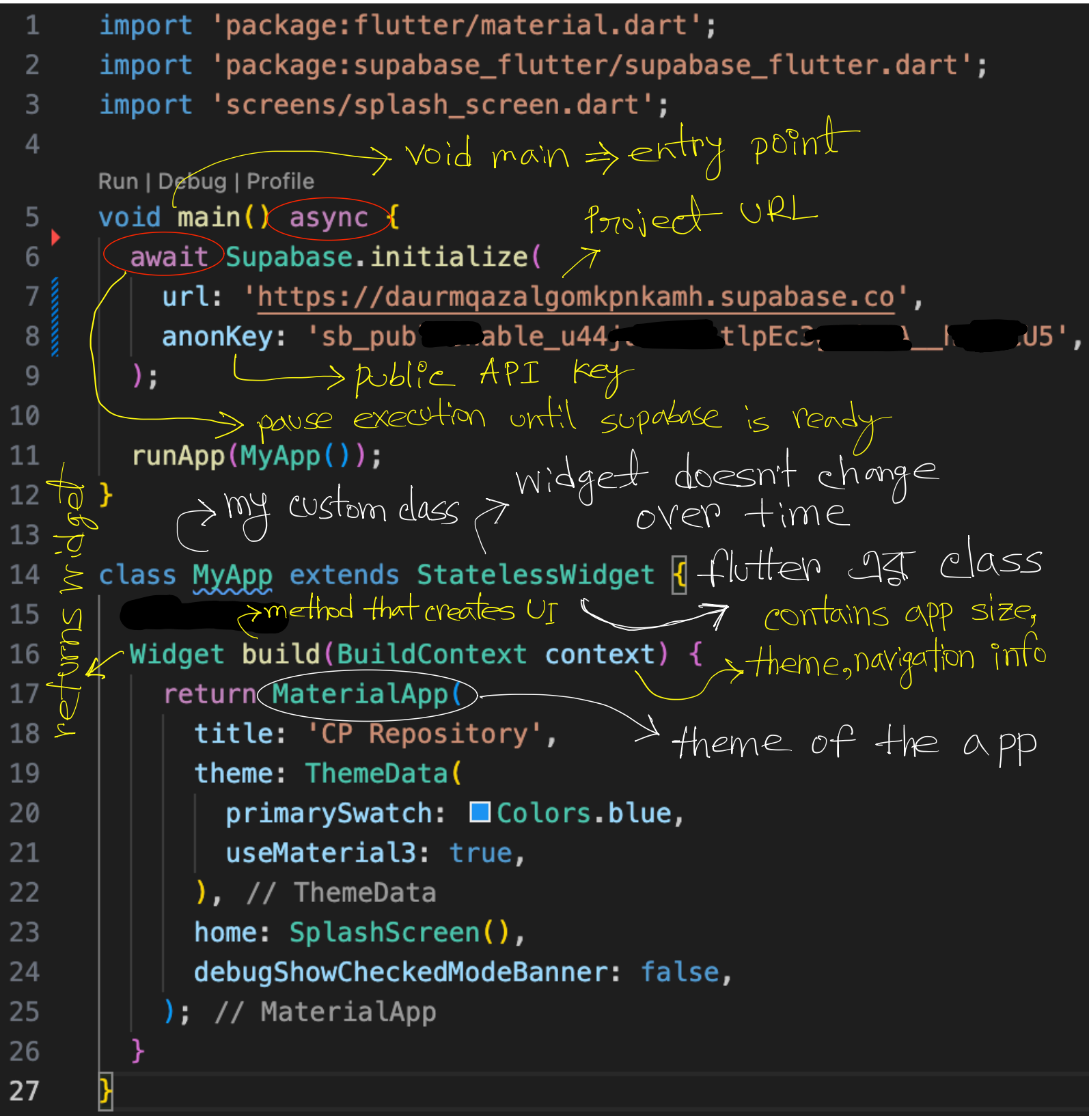
EASY

EASY

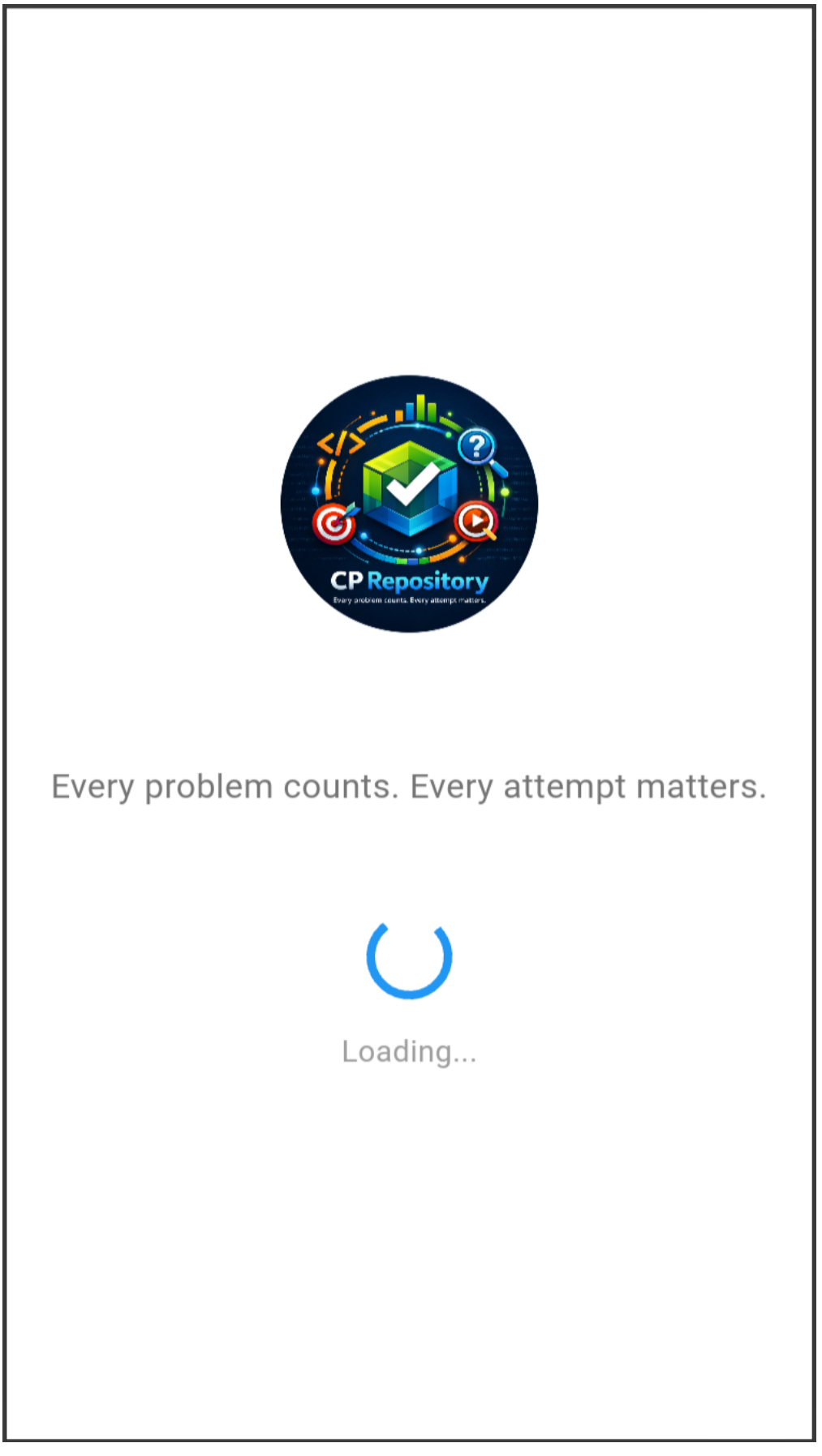
main.dart

Q: Why async on main()?

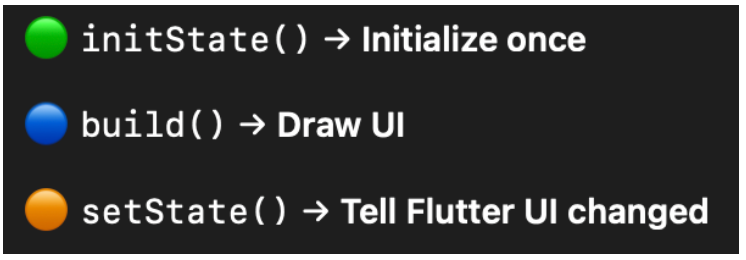
A: Because Supabase.initialize() returns a Future. We need await to wait for it to complete before runApp().



splash_screen.dart



remember this —




```dart
13    void initState() {
14      super.initState();
15      checkAuth();
16    }
```

→ call parent class

initState() is used for one-time initialization like auth checks and runs only once before the first build(), while setState() is meant to update data and rebuild UI multiple times.
Using setState() for initialization is wrong because it can't be called before the first build, causes unnecessary rebuilds, is bad for async logic, and may create infinite rebuild loops; splash auth needs no UI rebuild and navigation happens only once.

```dart
1   import 'package:flutter/material.dart';
2   import '../database.dart';
3   import 'login_screen.dart';
4   import 'home_screen.dart';
5
6   class SplashScreen extends StatefulWidget {
7     @override
8     SplashScreenState createState() => _SplashScreenState();
9   }
10
11  class _SplashScreenState extends State<SplashScreen> {
12    @override
13    void initState() {
14      super.initState();
15      checkAuth();
16    }
```

BTW, What is @override?

```
class A {
  void show() {
    print("A");
  }
}

class B extends A {
  void show() {
    print("B");
  }
}

void main() {
  A obj = B();
  obj.show();
}
```

Output: B

Child class method gets preference over parent class method for the same object → overriding

Method	Has Default?	What You Do
`createState()`	❌ NO	CREATE from scratch
`initState()`	✅ YES (empty)	EXTEND existing
`build()`	❌ NO	CREATE from scratch

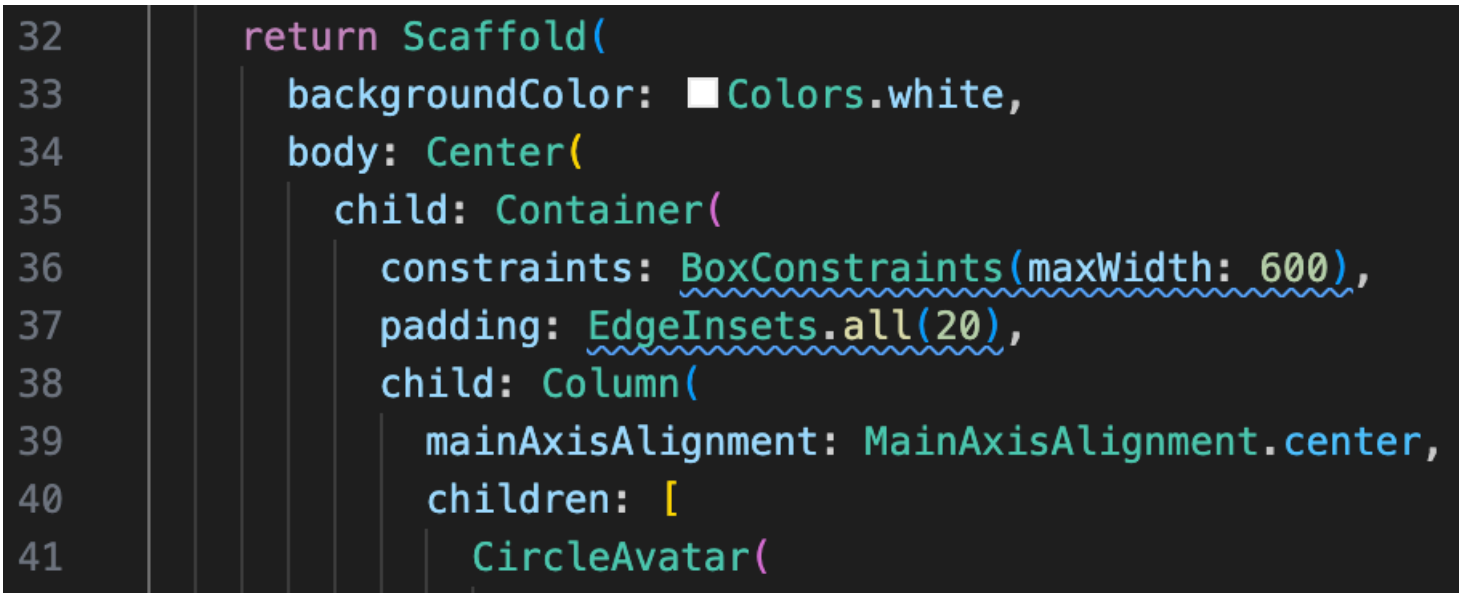
```
6   class SplashScreen extends StatefulWidget
7     @override
8     _SplashScreenState createState() => _Spl
9   }
```

@override here means — I'm implementing this abstract (non-existent) method.

```dart
29    Widget build(BuildContext context) {
30      final isSmallScreen = MediaQuery.of(context).size.width < 600;
```

build() is a framework-defined method from the Widget / State contract. Flutter calls this method automatically whenever it needs to draw or redraw the UI.

isSmallScreen checks the screen width to decide whether the device is small (phone) or large (tablet/desktop); if the width is less than 600 pixels, it treats it as a small screen.



```
32      return Scaffold(
33        backgroundColor: Colors.white,
34        body: Center(
35          child: Container(
36            constraints: BoxConstraints(maxWidth: 600),
37            padding: EdgeInsets.all(20),
38            child: Column(
39              mainAxisAlignment: MainAxisAlignment.center,
40              children: [
41                CircleAvatar(
```

Scaffold – Provides the basic screen structure, background color, and supports material design features like app bars, drawers, and snackbars.

Center – Centers its child widget both vertically and horizontally on the screen.

Container – A flexible box widget used to style and position content. You can add padding, margin, size constraints, decoration, and background color. Here, it limits the width (maxWidth: 600) and adds internal padding around its content.

Column – Arranges its children vertically in order. Allows alignment like mainAxisAlignment and crossAxisAlignment.

CircleAvatar – Displays a circular image or icon, commonly used for logos or profile pictures, with customizable size, background color, and optional image.

child – Refers to a single widget inside a parent widget (like Container or Center). Only one widget can be assigned.

children – Refers to a list of widgets inside a parent widget (like Column or Row). Multiple widgets can be arranged in order.

```dart
import 'package:flutter/material.dart';
import '../database.dart'; → to access isLoggedIn()
import 'login_screen.dart';
import 'home_screen.dart';

class SplashScreen extends StatefulWidget {
  @override
  _SplashScreenState createState() => _SplashScreenState();
}

class _SplashScreenState extends State<SplashScreen> {
  @override
  void initState() {
    super.initState();
    checkAuth(); → লগিন আছে কী-না।
  }
```

navigates (here, loading)

private class

```dart
Future<void> checkAuth() async {
  await Future.delayed(Duration(seconds: 2));

  if (isLoggedIn()) {
    Navigator.pushReplacement(context, MaterialPageRoute(builder: (_) => HomeScreen()));
  } else {
    Navigator.pushReplacement(context, MaterialPageRoute(builder: (_) => LoginScreen()));
  }
}
```

Widget location

Replace current screen

↳ manages screen stack ↳ trasition animation

Widget ⇒ Declaration
State ⇒ Logic

} StatefulWidget & State<T> separates

☆ Flutter UI automatically update হয় না। তাই variable update হলেও UI সেটা বুঝতে পারে না। এজন্য আমরা setState() ব্যবহার করি যেন স্ক্রিনে স্ক্রিনে Reload (UI upd.) হয়।

please turn over...

☆ যেহেতু StatefulWidget immutable, এবং আমাদের লজিক ও ডেটা প্রতিনিয়ত পরিবর্তনশীল, সেজন্য (changes, learns, grows) state এর ব্যবহার বাঞ্ছনীয়।

☆ তাহলে আমরা কেন সরাসরি state declare করে দিচ্ছি না?

কারণ state নিজে থেকে exist করতে পারে না। It is always tied to a widget. Because Flutter needs a widget tree to know whrer to place it on the screen.

```
Widget tree
 └─> StatefulWidget (immutable shell)
       └─> State (mutable object storing variables & logic)
```

সহজ বাংলায় (ইংলিশে),
Every state object is created by a StatefulWidget, which acts like a shell or container.

```dart
class _SplashScreenState extends State<Spla
  @override
  void initState() {
    super.initState();
    checkAuth();
  }
```

@override here means——
We are changing flutters built-in initState funtion and modifing it by @override, where calling the parent function (supe.initState), then add our checkAuth().

```dart
  @override
  Widget build(BuildConte
    return Scaffold(
      backgroundColor:
```

For build(),
We're providing our own complete implementation of the build() method using @override, since Flutter's build() is abstract (has no default).

In summery,
For StatefulWidget, State, and Widget, there is a backend where all functions are generated by Flutter that doesn't completely show in our code. When we make changes through extends, we're working inside these built-in classes. That's the purpose of using @override.

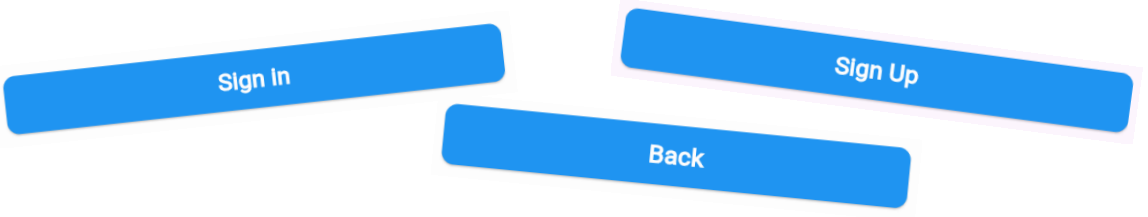
```dart
  @override
  Widget build(BuildContext context) {
    final isSmallScreen = MediaQuery.of(context).size.width < 600;

    return Scaffold(
      backgroundColor: Colors.white,
      body: Center(
        child: Container(
          constraints: BoxConstraints(maxWidth: 600),
          padding: EdgeInsets.all(20),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              CircleAvatar(
                radius: isSmallScreen ? 60 : 80,
                backgroundColor: Colors.blue.shade50,
                backgroundImage: AssetImage('assets/images/logo.png'),
              ), // CircleAvatar

              SizedBox(height: 60),

              Text(
                'Every problem counts. Every attempt matters.',
                style: TextStyle(
                  fontSize: isSmallScreen ? 16 : 18,
                  color: Colors.grey.shade600,
                ), // TextStyle
                textAlign: TextAlign.center,
              ), // Text

              SizedBox(height: 50),

              Column(
                children: [
                  CircularProgressIndicator(color: Colors.blue),
                  SizedBox(height: 16),
                  Text(
                    'Loading...',
                    style: TextStyle(color: Colors.grey.shade500),
                  ), // Text
                ],
              ), // Column
            ],
          ), // Column
        ), // Container
      ), // Center
    ); // Scaffold
  }
}
```

simple_button.dart

Login Screen এ এরকম কোড দেখি —

```
121        SimpleButton(text: 'Sign in', onPressed: doLogin, loading: isLoading),
```

creates a full-width button that shows “Sign in” when isLoading is false and runs doLogin when tapped; when isLoading is true, it shows a spinner and is disabled to prevent multiple taps.




```dart
import 'package:flutter/material.dart';

class SimpleButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;
  final bool loading;

  const SimpleButton({
    required this.text,
    required this.onPressed,
    this.loading = false,
  });

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: double.infinity,
      child: ElevatedButton(
        onPressed: loading ? null : onPressed,
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.blue,
          foregroundColor: Colors.white,
          padding: EdgeInsets.symmetric(vertical: 16),
          shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(8)),
        ),
        child: loading
            ? SizedBox(height: 20, width: 20, child: CircularProgressIndicator(strokeWidth: 2, color: Colors.white))
            : Text(text, style: TextStyle(fontSize: 16, fontWeight: FontWeight.w600)),
      ), // ElevatedButton
    ); // SizedBox
  }
}
```

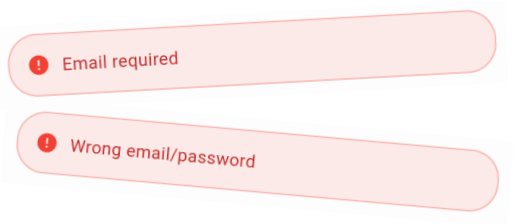
→ ফাংশন কল করবে

ElevatedButton is the widget that actually creates the button you see on the screen.

The child property sets the button content: if loading is true, it shows a small white spinner inside a 20×20 box; if false, it shows the button text with font size 16 and medium-bold weight.

message_widget.dart

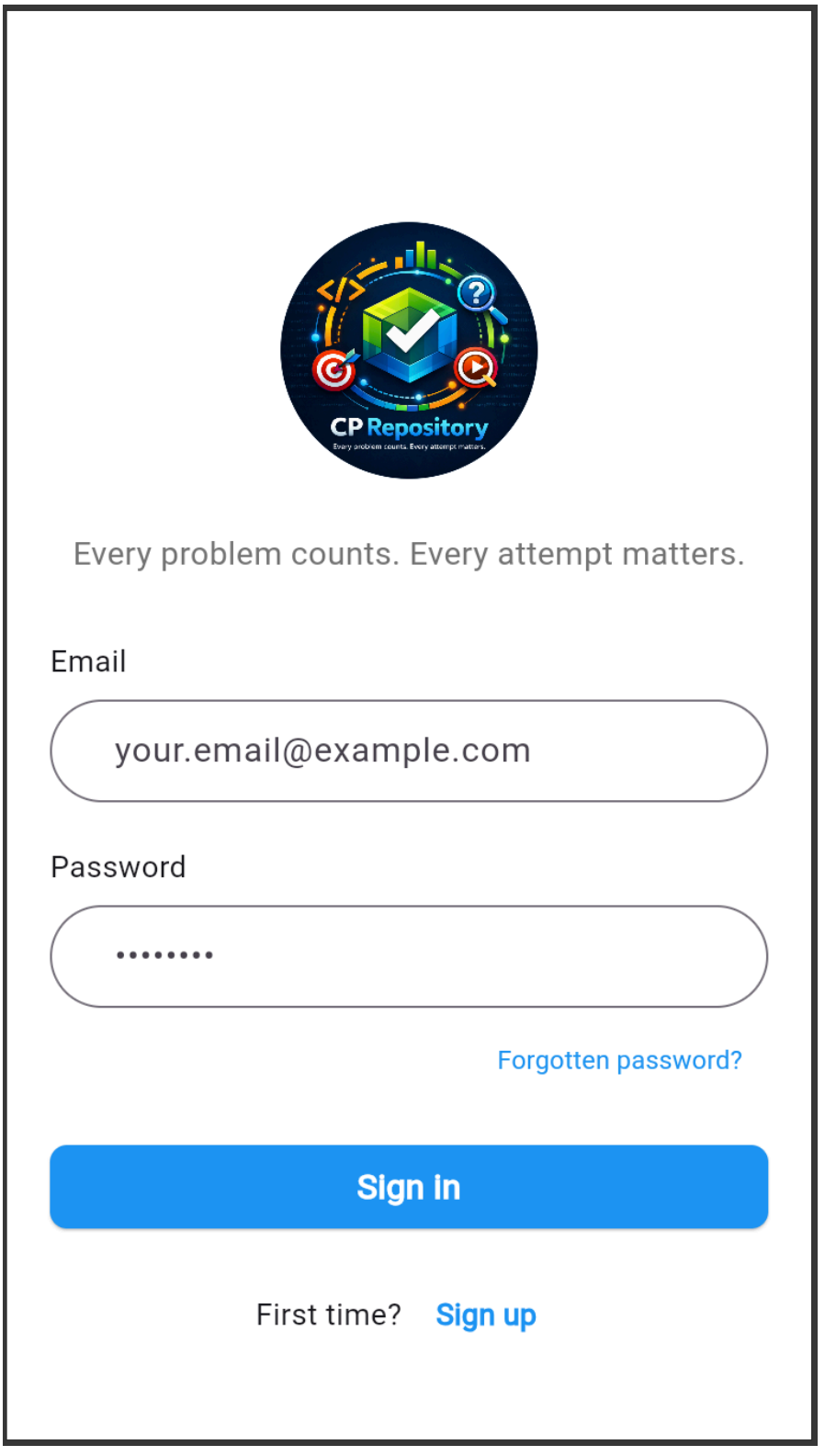
```dart
import 'package:flutter/material.dart';

class MessageWidget extends StatelessWidget {
  final String message;
  final bool isError;
  final IconData icon;
  final Color color;

  const MessageWidget({
    required this.message,
    this.isError = true,
    this.icon = Icons.error,
    this.color = Colors.red,
  });

  Color _backgroundColor() {
    if (isError) return Colors.red.withOpacity(0.1);
    return color.withOpacity(0.1);
  }

  Color _borderColor() {
    if (isError) return Colors.red.withOpacity(0.3);
    return color.withOpacity(0.3);
  }

  Color _iconColor() {
    if (isError) return Colors.red;
    return color;
  }

  Color _textColor() {
    if (isError) return Colors.red.shade800;
    if (color == Colors.green) return Colors.green.shade800;
    if (color == Colors.orange) return Colors.orange.shade800;
    if (color == Colors.blue) return Colors.blue.shade800;
    return color;
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(12),
      margin: EdgeInsets.only(bottom: 16),
      decoration: BoxDecoration(
        color: _backgroundColor(),
        borderRadius: BorderRadius.circular(20),
        border: Border.all(color: _borderColor()),
      ),
      child: Row(
        children: [
          Icon(icon, color: _iconColor(), size: 16),
          SizedBox(width: 8),
          Expanded(
            child: Text(
              message,
              style: TextStyle(color: _textColor(), fontSize: 12),
            ),
          ),
        ],
      ),
    );
  }
}
```



When calling MessageWidget, if colors/ icons aren't provided, it uses defaults (red for errors, Icons.error icon).

The Expanded widget makes the text fill all available horizontal space, preventing overflow while keeping the icon fixed on the left side for clean layout. If text exceeds space, it automatically wraps to multiple lines due to the Expanded widget's flexible nature and Flutter's automatic text wrapping behavior.

login_screen().dart



Before moving forward, we need a clear idea about setState(). setState() rebuilds the UI by calling the build() method again. The UI updates based on the new state values.

```dart
Future<void> doLogin() async {
  if (emailController.text.isEmpty) {
    setState(() => errorMsg = 'Email required');
    return;
  }
  if (!emailController.text.contains('@')) {
    setState(() => errorMsg = 'Valid email needed');
    return;
  }
  if (passwordController.text.isEmpty) {
    setState(() => errorMsg = 'Password required');
    return;
  }

  setState(() {
    isLoading = true;
    errorMsg = '';
  });

  try {
    await login(emailController.text.trim(), passwordController.text.trim());
    Navigator.pushReplacement(context, MaterialPageRoute(builder: (_) => HomeScreen()));
  } catch (e) {
    String msg = e.toString();
    if (msg.contains('Invalid login')) msg = 'Wrong email/password';
    setState(() => errorMsg = msg);
  } finally {
    setState(() => isLoading = false);
  }
}
```

setState() before the try block means all manual validations (email and password checks) have passed, and the loading spinner is shown.

After that, the code enters the try–catch–finally block.
If authentication succeeds, the user is navigated to the Home screen.
If authentication fails, setState() inside catch updates and shows the error message.

The finally block always executes, no matter whether login succeeds or fails.
If authentication fails, it turns off the loading spinner after showing the error.
If navigation happens, finally still runs because navigation does not immediately destroy the current widget, so the loading state is safely reset.

```dart
import 'package:flutter/material.dart';
import '../database.dart';
import '../widgets/simple_button.dart';
import '../widgets/message_widget.dart';
import 'register_screen.dart';
import 'home_screen.dart';
import 'forgot_password_screen.dart';

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final emailController = TextEditingController();
  final passwordController = TextEditingController();
  bool isLoading = false;
  String errorMsg = '';

  Future<void> doLogin() async {
    if (emailController.text.isEmpty) {
      setState(() => errorMsg = 'Email required');
      return;
    }
    if (!emailController.text.contains('@')) {
      setState(() => errorMsg = 'Valid email needed');
      return;
    }
    if (passwordController.text.isEmpty) {
      setState(() => errorMsg = 'Password required');
      return;
    }

    setState(() {
      isLoading = true;
      errorMsg = '';
    });

    try {
      await login(emailController.text.trim(), passwordController.text.trim());
      Navigator.pushReplacement(context, MaterialPageRoute(builder: (_) => HomeScreen()));
    } catch (e) {
      String msg = e.toString();
      if (msg.contains('Invalid login')) msg = 'Wrong email/password';
      setState(() => errorMsg = msg);
    } finally {
      setState(() => isLoading = false);
    }
  }
```

Why stateful?

- Email text changes
- Password text changes
- Loading state changes
- Error message changes

Let's differentiate between SingleChildScrollView vs Expanded

SingleChildScrollView handles vertical overflow by allowing the whole screen to scroll, while

Expanded handles horizontal space in a Row, constraining text to prevent overflow.

Together, they ensure the UI is safe, responsive, and scrollable in both directions.

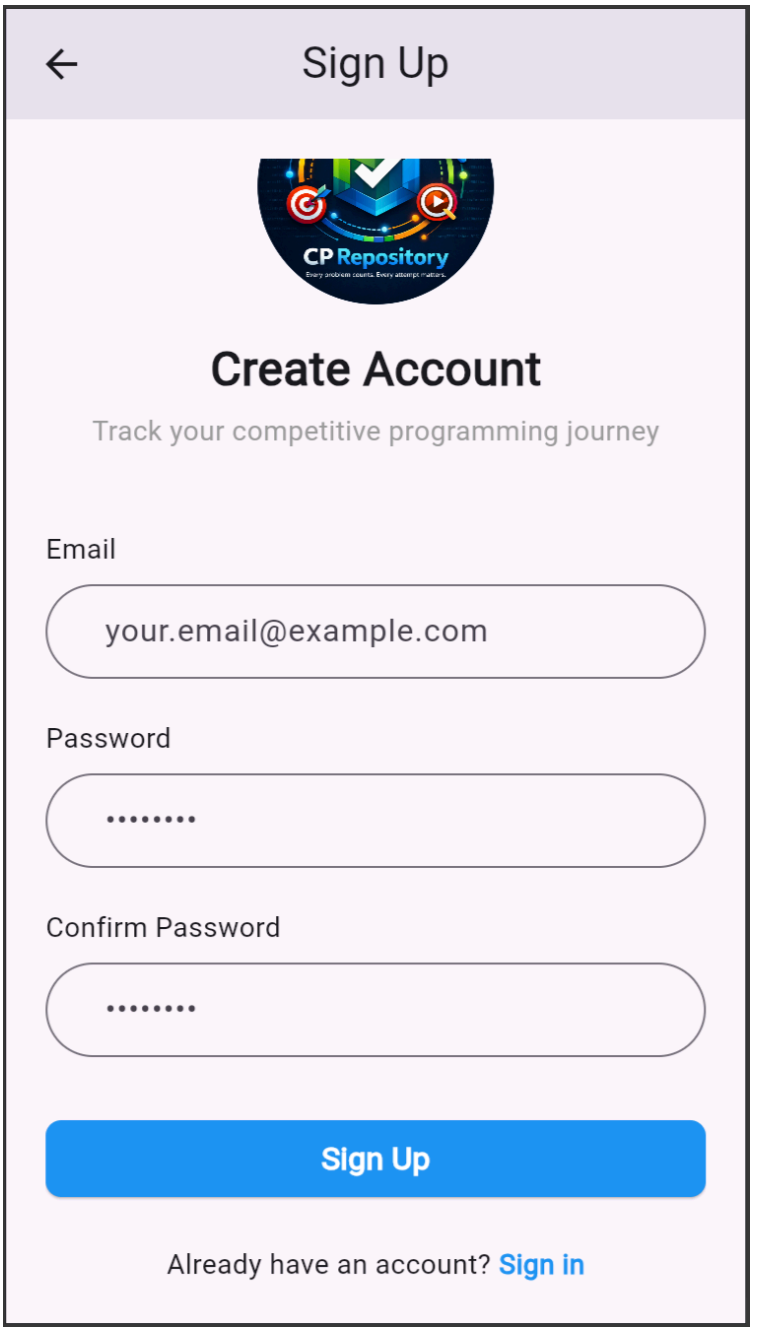
```dart
  @override
  Widget build(BuildContext context) {
    final isSmallScreen = MediaQuery.of(context).size.width < 600;

    return Scaffold(
      backgroundColor: Colors.white,
      appBar: AppBar(backgroundColor: Colors.white),
      body: Center(
        child: Container(
          constraints: BoxConstraints(maxWidth: 500),
          padding: EdgeInsets.all(20),
          child: SingleChildScrollView(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                CircleAvatar(
                  radius: isSmallScreen ? 60 : 80,
                  backgroundColor: Colors.blue.shade50,
                  backgroundImage: AssetImage('assets/images/logo.png'),
                ), // CircleAvatar
                SizedBox(height: 24),
                Text('Every problem counts. Every attempt matters.',
                  style: TextStyle(fontSize: isSmallScreen ? 15 : 18,
                  color: Colors.grey.shade600)), // TextStyle // Text
                SizedBox(height: 30),

                if (errorMsg.isNotEmpty)
                  MessageWidget(message: errorMsg, isError: true),

                Column(
                  crossAxisAlignment: CrossAxisAlignment.start,
                  children: [
                    Text('Email', style: TextStyle(fontWeight: FontWeight.w500)),
                    SizedBox(height: 8),
                    TextFormField(
                      controller: emailController,
                      keyboardType: TextInputType.emailAddress,
                      decoration: InputDecoration(
                        hintText: 'your.email@example.com',
                        border: OutlineInputBorder(borderRadius: BorderRadius.circular(30)),
                        contentPadding: EdgeInsets.symmetric(horizontal: 26, vertical: 14),
                      ), // InputDecoration
                    ), // TextFormField
                    SizedBox(height: 20),

                    Text('Password', style: TextStyle(fontWeight: FontWeight.w500)),
                    SizedBox(height: 8),
                    TextFormField(
                      controller: passwordController,
                      obscureText: true,
                      decoration: InputDecoration(
                        hintText: '••••••••',
                        border: OutlineInputBorder(borderRadius: BorderRadius.circular(30)),
                        contentPadding: EdgeInsets.symmetric(horizontal: 26, vertical: 14),
                      ), // InputDecoration
                    ), // TextFormField
```

```dart
107                  SizedBox(height: 8),
108
109                  Align(
110                    alignment: Alignment.centerRight,
111                    child: TextButton(
112                      onPressed: () => Navigator.push(context,
113                        MaterialPageRoute(builder: (_) => ForgotPasswordScreen())),
114                      child: Text('Forgotten password?',
115                        style: TextStyle(color: Colors.blue,
116                        fontSize: isSmallScreen ? 12 : 14)), // TextStyle // Text
117                    ), // TextButton
118                  ), // Align
119                  SizedBox(height: 24),
120
121                  SimpleButton(text: 'Sign in', onPressed: doLogin, loading: isLoading),
122                  SizedBox(height: 24),
123
124                  Row(
125                    mainAxisAlignment: MainAxisAlignment.center,
126                    children: [
127                      Text('First time?'),
128                      SizedBox(width: 4),
129                      TextButton(
130                        onPressed: () => Navigator.push(context,
131                          MaterialPageRoute(builder: (_) => RegisterScreen())),
132                        child: Text('Sign up',
133                          style: TextStyle(color: Colors.blue, fontWeight: FontWeight.w600)), // Text
134                      ), // TextButton
135                    ],
136                  ), // Row
137                ],
138              ), // Column → ডিফল্ট কলামের ভেতরে মতোভাব।
139            ],
140          ), // Column
141        ), // SingleChildScrollView
142      ), // Container
143    ), // Center
144  ); // Scaffold
145  }
146 }
```

Center → centers its child BOTH horizontally AND vertically.
Container → padding + max width.
SingleChildScrollView → avoids overflow on small screens.
Column → vertical layout.

Align is used here to push the “Forgotten password?” button to the right edge of its parent, ensuring proper positioning on all screen sizes, which a simple SizedBox cannot do reliably.

register_screen.dart

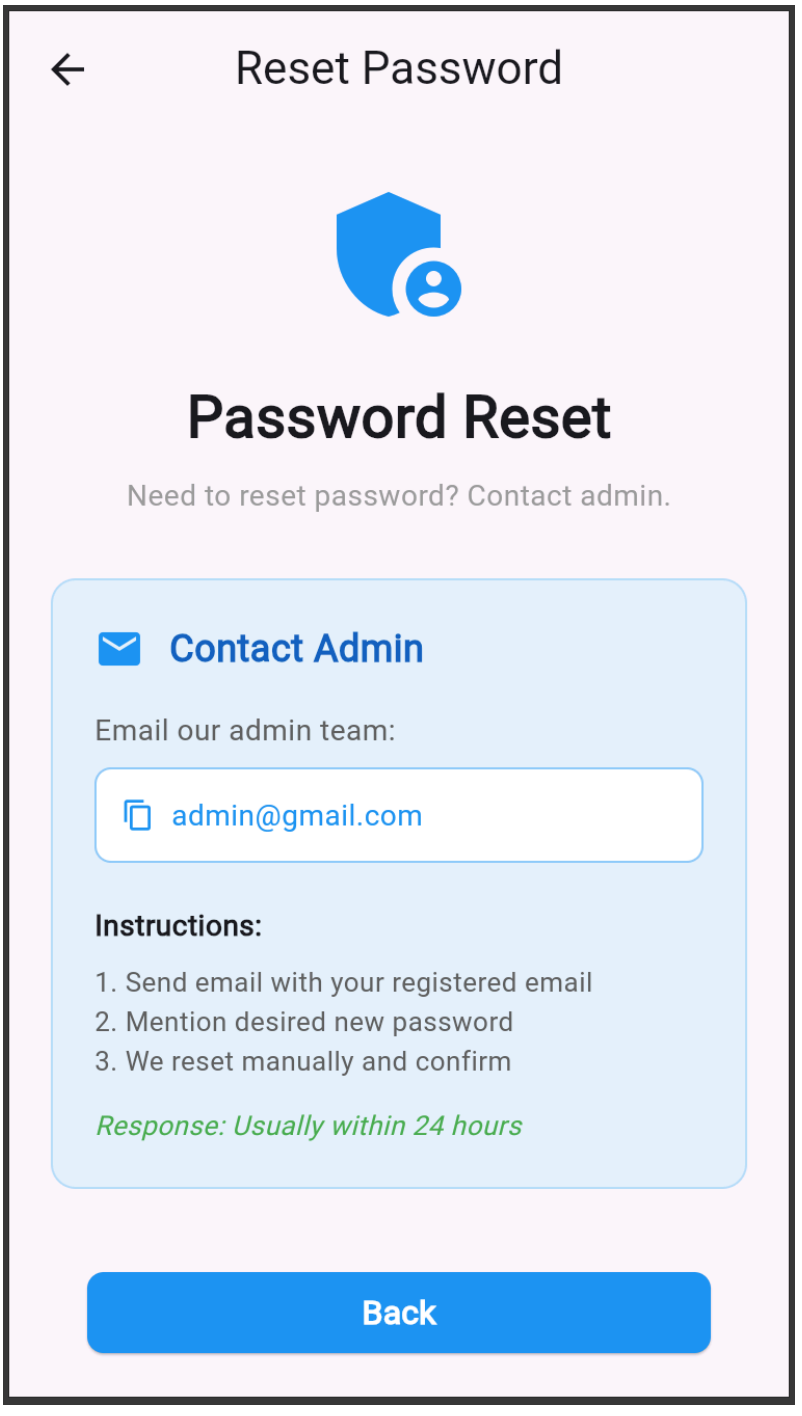


The code in register_screen.dart is similar to the login screen's code. To avoid repetition, no explanation of this file is included. Just one info:

Why the back arrow is always on the LEFT?

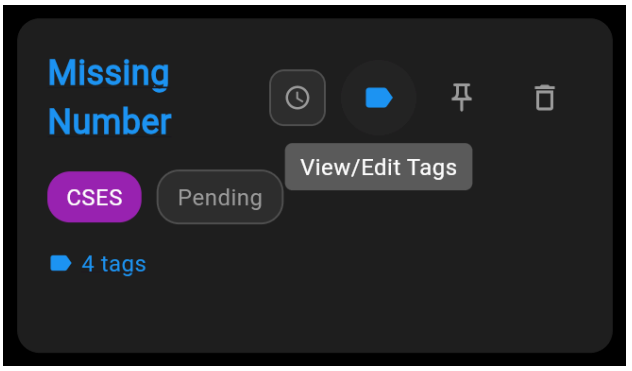
Because Material Design + AppBar layout always on the start (left) side.

forgot_password_screen.dart



easy, easy---

problem_card.dart



ProblemCard is complex because it's doing MANY things:

Two responsive layouts - Mobile card vs desktop row

Interactive features - 4 different click actions (status, tags, pin, delete)

Dynamic styling - Colors change based on status/platform/pinned state

URL handling - Clickable titles that open in browser

Multiple visual states - Different looks for solved/attempt/pending

Platform detection - Auto-colors for 7+ CP platforms

Nested widgets - 4 helper classes inside one file

Conditional rendering - Shows/hides elements based on state

It's essentially a "mini-app" within your app - the most feature-rich component that handles display AND interactions for each problem. The complexity comes from packing many features into a reusable, responsive component.

মাথা ঘুরে গেছে তাই ইগ্নোর করলাম :)

(btw, added text notes)

This file is purely about presentation and interaction, not data ownership. The ProblemCard widget does not decide anything by itself; it only displays what it is told and reports user actions upward. That separation is very intentional. The card never mutates data, never talks to the database directly for changes, and never assumes global state. It behaves like a “dumb but interactive” component, which makes it reusable and safe.

The widget supports two visual personalities: a compact card layout and a wide row layout. Instead of creating two separate widgets, a single flag decides which layout to render. The build method acts as a router that delegates UI responsibility to the appropriate private builder. This keeps logic centralized while preventing the main build method from becoming unreadable.

Every interactive element in the card is callback-driven. Tapping status, tags, delete, or pin does not trigger logic inside the card; it simply notifies the parent that something happened. This means the card is unaware of navigation, dialogs, or database updates. The parent screen stays in control of behavior, while the card focuses only on user affordances and visual feedback.

The title section shows how UI subtly communicates capability. A problem with a valid URL becomes visually clickable through color and underline, and only then does tapping attempt to open the link. This avoids accidental actions and teaches the user what is interactive without explicit instructions.

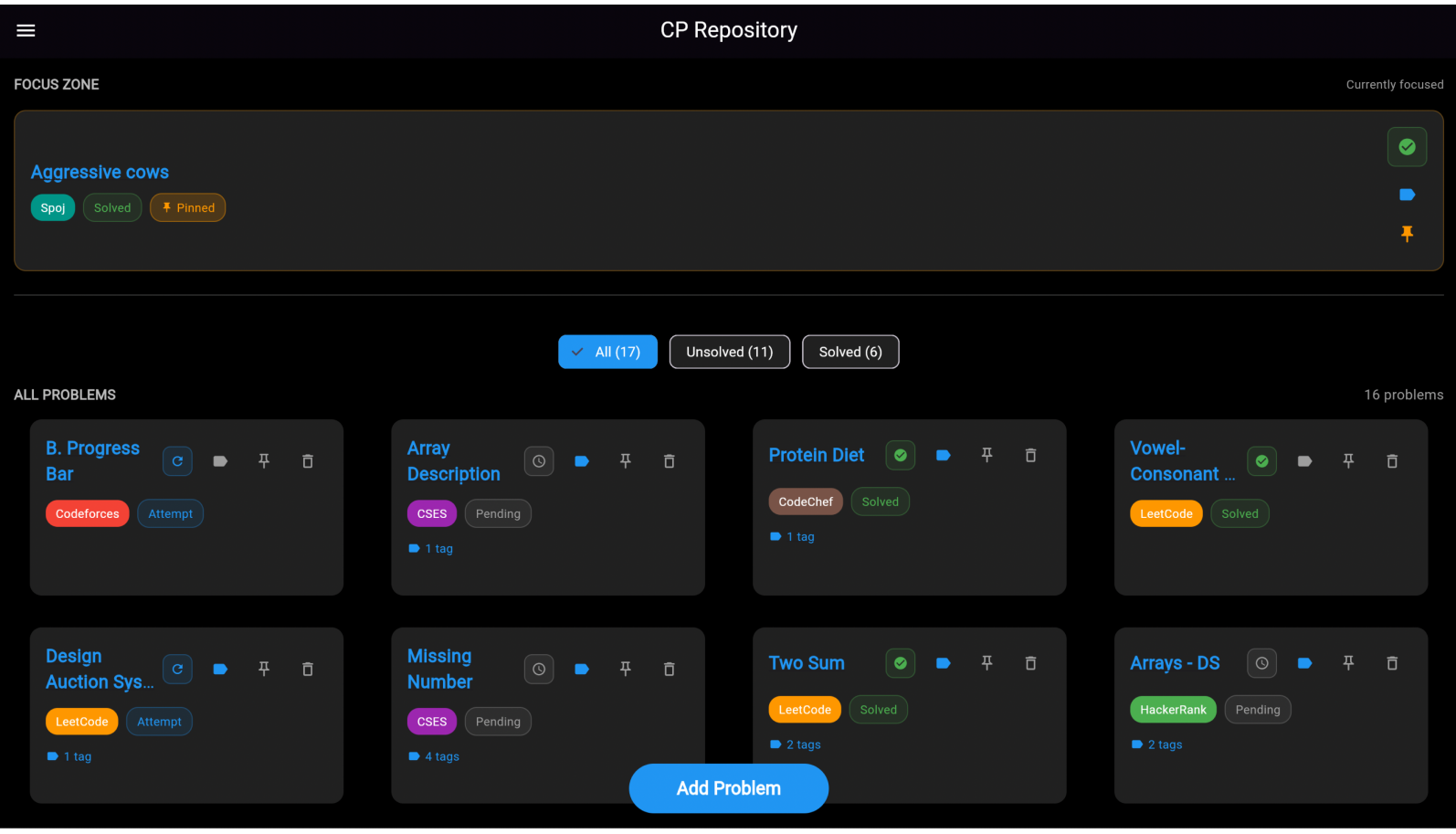
Status, platform, and pinned state are each represented as small, self-contained widgets. These components encode meaning visually through color, icon choice, and shape instead of text explanations. Status icons are clickable and intentionally larger to encourage interaction, while chips are passive indicators. This hierarchy reduces cognitive load and keeps the card scannable even with many problems on screen.

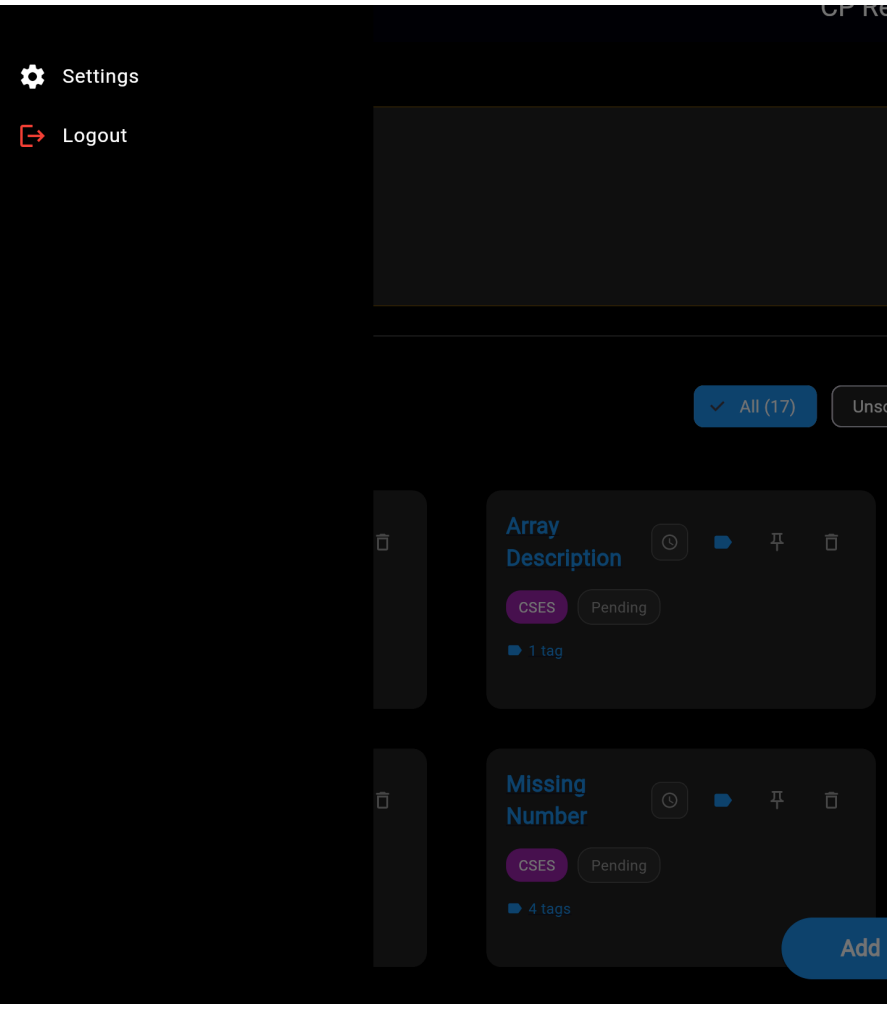
Pinned behavior is handled visually, not logically, inside this file. The card only reflects whether it is pinned and exposes a toggle action. Any special treatment, such as rendering pinned items elsewhere or enforcing single-pin rules, is handled outside. This prevents UI components from secretly controlling application rules.

Tag visibility is progressive rather than noisy. Tags are not dumped into the card; instead, their presence is summarized and expandable. This balances information density while still making tags discoverable. The same tag action works whether tags exist or not, which keeps interaction consistent.

Overall, this file demonstrates a strong boundary between UI and control flow. It treats widgets as instruments, not decision-makers. The result is a component that is easy to test, easy to redesign, and hard to misuse. It’s the kind of file that doesn’t look impressive at first glance, but quietly carries a lot of architectural discipline.

home_screen.dart





The HomeScreen is a Flutter stateful widget that acts as the main dashboard for managing competitive programming problems. It maintains a list of problems fetched from the database along with statistics for total, pending, attempted, and solved problems, and treats the database as the source of truth. From the moment the screen starts, it immediately fetches all problems and stats, and the widget only keeps a temporary copy of the data to render the UI. Any user action, such as changing status, adding tags, deleting a problem, or adding a new one, first updates the database and then refreshes the local state, ensuring that the UI remains stable and predictable. The screen supports filtering by status using FilterChips, allowing users to view all problems, only unsolved problems, or only solved problems. A pinned problem, if any, is displayed at the top in a “Focus Zone” to draw attention, and the rest of the problems are shown below. Filtered lists are never stored; they are derived from the main list on demand, and pinning works by tracking a single problem ID and rendering it separately.

The layout uses a Stack deliberately to allow overlaying the “Add Problem” floating button on top of the scrollable problem list without affecting the rest of the layout, avoiding the need for extra padding or a FloatingActionButton. The pinned problem and non-pinned problem lists are rendered separately; the pinned problem uses a row layout on wider screens for a compact horizontal display and skips row layout on narrow screens to avoid cramped content. Similarly, the main problem list switches between a ListView for small screens and a GridView for larger screens, making the UI responsive without scattering layout conditionals across multiple widgets.

Dialogs, including the TagDialog, are treated as temporary decision points that collect user input without owning state. The TagDialog keeps a local copy of tags, so canceling never causes side effects, and only saving commits changes to the database. All interactions trigger state updates and database calls while providing immediate feedback through SnackBars or dialogs. Overall, the file centralizes logic, derives the UI from state, and re-fetches data after mutations, prioritizing clarity, control, and resilience over clever optimizations, resulting in a modular, responsive, and programmer-friendly interface.

add_problem_screen.dart

The AddProblemScreen is a Flutter stateful widget that provides a user interface for adding new programming problems to a database. It includes text fields for entering the problem title, URL, and platform, all of which are required, along with an optional input for adding tags. Users can type a tag and add it to a list, which is displayed dynamically using chips that can be removed individually. The screen also features a status selector implemented with ChoiceChips, allowing users to mark the problem as Pending, Attempt, or Solved, each with a distinct color for visual clarity. The save function validates the required fields, constructs a Problem object, and saves it to the database, providing feedback through SnackBars and showing a loading state while the operation completes. The layout is designed for a dark theme with responsive behavior, using a scrollable container and a maximum width to ensure usability on different screen sizes. Overall, the file efficiently handles user input, state management, and feedback, offering an interactive and visually organized form for problem tracking.

profile_edit_screen.dart

easy.

settings_screen.dart

easy.

welcome_screen.dart

easy.