Estructura Básica de una Aplicación Flutter

# **Fundamentos Flutter**

## **1.1 Widgets (componentes Basicos)**

Los widgets son los componentes fundamentales de cualquier aplicación Flutter. Vamos a explorarlos en detalle, basándonos en el código que has compartido.

### Concepto Básico de Widgets

**Definición**: Un widget es una descripción inmutable de parte de una interfaz de usuario. En Flutter, todo es un widget: desde elementos simples como texto o botones, hasta estructuras complejas como pantallas completas.

Los widgets son los elementos fundamentales de construcción en Flutter. Representan:

Componentes visuales (botones, texto)

Estructuras de layout (columnas, filas)

Pantallas completas

**Características clave**:

Son inmutables (una vez creados no cambian)

Se componen jerárquicamente

Todo en Flutter es un widget

### Tipos de Widgets en el Código

#### 1. Widgets de Layout (Diseño)

#### **MaterialApp**:

**MaterialApp(**

**title: 'App de Cobros',**

**debugShowCheckedModeBanner: false,**

**theme: AppTheme.lightTheme,**

**home: StreamBuilder<User?>(**

**// ...**

**),**

**)**

Proporciona estructura Material Design

Configuración global de la app

**Función**: Proporciona la estructura básica de una aplicación Material Design

**Propiedades clave**:

title: Nombre de la app (usado por el sistema operativo)

theme: Define los colores, tipografías y estilos globales

home: Widget principal que se muestra al iniciar la app

#### **Scaffold** (en \_buildLoadingScreen y \_buildErrorScreen):

**Scaffold(**

**body: Center(**

**child: Column(**

**// ...**

**),**

**),**

**)**

**Función**: Proporciona una estructura básica de página con AppBar, Body, etc.

**Propiedades**:

body: Contenido principal de la pantalla

### 2. Widgets de Estado

#### **StatefulWidget** (Ej:AuthWrapper):

**class AuthWrapper extends StatefulWidget {**

**const AuthWrapper({super.key});**

**@override**

**State<AuthWrapper> createState() => \_AuthWrapperState();**

**}**

**Características**:

Mantiene estado mutable que puede cambiar durante la vida del widget

Compuesto por dos clases: la propia StatefulWidget y su State

#### State (\_AuthWrapperState):

**class \_AuthWrapperState extends State<AuthWrapper> with WidgetsBindingObserver {**

**DateTime? \_lastVerificationTime;**

**bool \_isCheckingMembership = false;**

**// Métodos de estado...**

**}**

**Responsabilidades**:

Mantiene datos mutables (variables de instancia)

Implementa la lógica de negocio

Define el método build() que describe la UI

### 3. Widgets Funcionales

#### **StreamBuilder**:

**StreamBuilder<User?>(**

**stream: FirebaseAuth.instance.authStateChanges(),**

**builder: (context, authSnapshot) {**

**// ...**

**}**

**)**

**Función**: Reconstruye la UI cuando el stream emite nuevos datos

**Partes**:

stream: Fuente de datos asíncronos (en este caso, cambios de autenticación)

builder: Función que construye la UI basada en el último dato del stream

#### **FutureBuilder**:

**FutureBuilder<Map<String, dynamic>?>(**

**future: Provider.of<UserService>(context).getCurrentUserData(),**

**builder: (context, userSnapshot) {**

**// ...**

**}**

**)**

**Función**: Similar a StreamBuilder pero para un único Future en lugar de un stream continuo

**Usado para**: Datos que se cargan una vez (como información de usuario)

### 4. Widgets de UI Básicos

#### **Column**:

**Column(**

**mainAxisAlignment: MainAxisAlignment.center,**

**children: [**

**const CircularProgressIndicator(),**

**const SizedBox(height: 20),**

**Text(message)**

**],**

**)**

**Función**: Organiza widgets hijos verticalmente

**Propiedades clave**:

mainAxisAlignment: Alineación vertical

children: Lista de widgets hijos

#### **Text**:

**Text('Error: $error', style: const TextStyle(color: Colors.red))**

**Función**: Muestra texto en pantalla

**Personalización**: Estilos, tamaños, colores mediante TextStyle

### Ciclo de Vida de los Widgets Stateful

En \_AuthWrapperState vemos varios métodos del ciclo de vida:

#### **initState()**:

**@override**

**void initState() {**

**super.initState();**

**WidgetsBinding.instance.addObserver(this);**

**\_lastVerificationTime = DateTime.now();**

**\_verifyInitialMembership();**

**}**

Se ejecuta una sola vez cuando el widget se inserta en el árbol

Lugar ideal para:

Inicializar variables

Suscribirse a streams

Registrar observadores

#### **didChangeAppLifecycleState()**:

**@override**

**void didChangeAppLifecycleState(AppLifecycleState state) {**

**if (state == AppLifecycleState.resumed) {**

**\_checkMembershipOnResume();**

**}**

**}**

Reacciona a cambios en el estado de la app (pausada, reanudada, etc.)

En este caso, verifica membresía cuando la app vuelve al primer plano

#### **dispose()**:

**@override**

**void dispose() {**

**WidgetsBinding.instance.removeObserver(this);**

**super.dispose();**

**}**

Se ejecuta cuando el widget se elimina permanentemente

Crucial para:

Cancelar suscripciones

Liberar recursos

Remover observadores

#### **build()**:

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**// ...**

**);**

**}**

Describe cómo mostrar el widget en función de su estado actual

Se llama automáticamente cuando:

El widget se inserta en el árbol

Se llama a setState()

Cambian las dependencias (en Provider, InheritedWidget, etc.)

### Composición de Widgets

El código muestra un excelente ejemplo de composición de widgets:

**MaterialApp**

**└─ StreamBuilder**

**└─ FutureBuilder**

**├─ MembershipScreen (si membresía inválida)**

**└─ MainScreen (si todo está bien)**

Esta anidación permite:

Separación de preocupaciones

Reutilización de componentes

Mantenimiento más sencillo

### Widgets Personalizados

El código incluye widgets personalizados como:

#### **AuthWrapper**:

Gestiona el estado global de autenticación

Coordina la lógica de verificación de membresía

Decide qué pantalla mostrar

#### **\_buildLoadingScreen** y **\_buildErrorScreen**:

Métodos que devuelven widgets para estados específicos

Ejemplo de cómo encapsular UI repetitiva

# 2. Importaciones (import)

**Las líneas que comienzan con import son para incluir funcionalidades de otros paquetes o archivos en nuestro código.**

## **1. Bibliotecas Principales de Flutter**

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

**Qué es: La biblioteca fundamental para construir interfaces de usuario en Flutter.**

### **Proporciona:**

**Todos los widgets básicos (Text, Button, Scaffold, AppBar, etc.)**

**Temas y estilos básicos**

**Sistema de navegación (rutas)**

**Gestión de layouts (Column, Row, Stack, etc.)**

**Esencial para cualquier aplicación Flutter**

## **2. Firebase Core y Autenticación**

**import 'package:firebase\_core/firebase\_core.dart';**

**Qué es: El paquete central necesario para usar cualquier servicio de Firebase.**

### **Proporciona:**

**Inicialización de Firebase (Firebase.initializeApp())**

## **Configuración básica de la conexión con Firebase**

**import 'package:firebase\_auth/firebase\_auth.dart';**

**Qué es: El paquete para autenticación con Firebase.**

### **Proporciona:**

**Autenticación con email/contraseña**

**Autenticación con Google, Facebook, etc.**

**Gestión del usuario actual (FirebaseAuth.instance.currentUser)**

**Stream de cambios de autenticación (authStateChanges())**

## **3. Utilidades del Sistema**

**import 'dart:io';**

**Qué es: Biblioteca de Dart para operaciones del sistema.**

**Proporciona:**

**Detección de plataforma (Platform.isAndroid, Platform.isWindows, etc.)**

**Operaciones con archivos y directorios**

**Funcionalidades específicas del sistema operativo**

## **4. Configuración de Firebase**

**import 'firebase\_options.dart';**

**Qué es: Archivo generado automáticamente por FlutterFire.**

### **Proporciona:**

**Configuración específica para cada plataforma (iOS, Android, Web)**

**Claves y configuraciones necesarias para inicializar Firebase**

**Se genera al ejecutar flutterfire configure**

## **5. Pantallas de la Aplicación**

**import 'screens/login\_screen.dart';**

**import 'screens/main\_screen.dart';**

**import 'screens/roles/owner/member\_ship\_screen.dart';**

**Qué son: Importaciones de archivos locales que contienen las pantallas de la aplicación.**

### **Proporcionan:**

**login\_screen.dart: Pantalla de inicio de sesión**

**main\_screen.dart: Pantalla principal después del login**

**member\_ship\_screen.dart: Pantalla de gestión de membresías**

## **6. Base de Datos Local**

**import 'package:sqflite\_common\_ffi/sqflite\_ffi.dart';**

**Qué es: Paquete para usar SQLite en Flutter, con soporte para escritorio.**

### **Proporciona:**

**Implementación de SQLite para Windows/Linux/Mac**

**Permite usar la misma API de SQLite en todas las plataformas**

**sqfliteFfiInit() inicializa el sistema para escritorio**

## **7. Servicios de la Aplicación**

**import 'services/user\_service.dart';**

**import 'services/auth\_service.dart';**

**Qué son: Clases propias que encapsulan lógica de negocio.**

### **Proporcionan:**

**user\_service.dart: Maneja operaciones relacionadas con usuarios**

**auth\_service.dart: Maneja lógica de autenticación y membresías**

## **8. Internacionalización**

**import 'package:intl/date\_symbol\_data\_local.dart';**

**Qué es: Paquete para internacionalización y formatos locales.**

### **Proporciona:**

**Formatos de fecha/hora según localización**

**Símbolos y convenciones regionales**

**initializeDateFormatting() carga los datos para un locale específico ('es\_CO' en este caso)**

## **9. Tema de la Aplicación**

**import '/utils/app\_theme.dart';**

**Qué es: Archivo local que define el tema visual de la app.**

### **Proporciona:**

**Colores, tipografías y estilos consistentes**

**Definición de AppTheme.lightTheme usado en MaterialApp**

## **10. Trabajo en Segundo Plano**

**import 'package:workmanager/workmanager.dart';**

**Qué es: Paquete para ejecutar tareas en background.**

### **Proporciona:**

**Ejecución de código periódicamente aunque la app esté cerrada**

**Workmanager().initialize() configura el manejador de tareas**

**registerPeriodicTask() programa tareas recurrentes**

## **11. Gestión de Estado**

**import 'package:provider/provider.dart';**

**Qué es: Implementación del patrón Provider para gestión de estado.**

### **Proporciona:**

**Provider y MultiProvider para inyección de dependencias**

**Provider.of(context) para acceder a los servicios desde los widgets**

**Sistema sencillo para manejar estado compartido**

**Concepto básico: Las importaciones son como incluir bibliotecas o herramientas que necesitamos usar en nuestro código.**

### **Resumen de Funcionalidades por Capas**

**UI/Core: material.dart, app\_theme.dart**

**Infraestructura: firebase\_core.dart, firebase\_auth.dart**

**Persistencia: sqflite\_ffi.dart, user\_service.dart**

**Lógica de negocio: auth\_service.dart**

**Utilidades: intl, dart:io**

**Background: workmanager.dart**

**Estado: provider.dart**

**Cada importación cumple un propósito específico y juntas forman la base para una aplicación robusta con autenticación, base de datos local, trabajo en background y una estructura bien organizada.**

# **2. Función Principal (main)**

**La función main() es el punto de entrada de toda aplicación Dart/Flutter.**

## Estructura Básica

**void main() async {**

**// 1. Configuración inicial**

**// 2. Inicialización de Firebase**

**// 3. Configuración adicional**

**// 4. Workmanager (solo mobile)**

**// 5. Ejecución de la app**

**}**

## 1. Configuración Inicial

**if (Platform.isWindows || Platform.isLinux || Platform.isMacOS) {**

**sqfliteFfiInit();**

**databaseFactory = databaseFactoryFfi;**

**}**

**WidgetsFlutterBinding.ensureInitialized();**

### Explicación detallada:

**Detección de plataforma: Platform.isWindows/Linux/MacOS**

**Verifica si la app se ejecuta en sistemas de escritorio**

**Importante porque SQLite requiere configuración especial en estas plataformas**

**Inicialización de SQLite FFI:**

**sqfliteFfiInit(): Prepara el sistema para usar SQLite via FFI (Foreign Function Interface)**

**databaseFactory = databaseFactoryFfi: Establece la fábrica de bases de datos para plataformas de escritorio**

**FFI permite llamar a código nativo (C/C++) desde Dart**

**WidgetsFlutterBinding.ensureInitialized():**

**Conecta el framework con el motor de Flutter**

**Obligatorio antes de usar cualquier widget de Flutter**

**Necesario para llamar a APIs asíncronas antes de runApp()**

## 2. Inicialización de Firebase

**try {**

**await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);**

**} catch (e) {**

**debugPrint('Error al inicializar Firebase: $e');**

**}**

## Desglose:

### Firebase.initializeApp():

Inicializa la conexión con los servicios de Firebase

Es asíncrono (await), por eso main() es async

### DefaultFirebaseOptions.currentPlatform:

Configuración generada automáticamente por FlutterFire

Contiene las claves y configuraciones específicas para cada plataforma

Se encuentra en firebase\_options.dart

### Manejo de errores:

Bloque try-catch para capturar errores de inicialización

debugPrint muestra el error en la consola de depuración

## 3. Configuración Adicional

**await initializeDateFormatting('es\_CO', null);**

#### Profundización:

##### initializeDateFormatting():

Del paquete intl

Carga los patrones de formato de fecha para un locale específico ('es\_CO' - español de Colombia)

Permite mostrar fechas localizadas correctamente

Ejemplo: "martes, 12 de octubre" en lugar de "Tuesday, October 12"

## 4. Configuración de Workmanager (solo mobile)

**if (Platform.isAndroid || Platform.isIOS) {**

**await Workmanager().initialize(callbackDispatcher, isInDebugMode: false);**

**await Workmanager().registerPeriodicTask(**

**'membershipCheck',**

**'membershipCheck',**

**frequency: const Duration(hours: 12),**

**constraints: Constraints(networkType: NetworkType.connected),**

**);**

**}**

### Análisis exhaustivo:

#### Condición plataforma móvil:

Solo se ejecuta en Android/iOS

Workmanager no es necesario en escritorio/web

#### Workmanager.initialize():

Configura el manejador de tareas en background

callbackDispatcher: Función que se ejecutará cuando se dispare la tarea

isInDebugMode: false: Desactiva logs detallados en producción

#### registerPeriodicTask():

Programa una tarea recurrente cada 12 horas

Parámetros:

'membershipCheck': Identificador único de la tarea

frequency: Intervalo de ejecución (mínimo 15 minutos en Android)

constraints: Restricciones (ej. requiere conexión a internet)

## 5. Ejecución de la Aplicación

**runApp(**

**MultiProvider(**

**providers: [**

**Provider<AuthService>(create: (\_) => AuthService()),**

**Provider<UserService>(create: (\_) => UserService()),**

**],**

**child: const AuthWrapper(),**

**),**

**);**

### Descomposición:

#### runApp():

Monta el widget raíz en el árbol de widgets

Inicia el ciclo de vida de la aplicación

#### MultiProvider:

Widget que provee múltiples instancias de servicios/proveedores

Patrón de inyección de dependencias

#### Providers registrados:

AuthService: Maneja autenticación y membresías

UserService: Maneja datos y operaciones de usuario

#### AuthWrapper:

Widget raíz personalizado

StatefulWidget que maneja el estado global de autenticación

Observa cambios en el ciclo de vida de la app

**Conceptos:**

**main(): Es la función que se ejecuta primero en cualquier aplicación Dart/Flutter.**

**async: Indica que la función contiene operaciones asíncronas (que toman tiempo).**

**runApp(): Función de Flutter que inicia la aplicación con el widget raíz (en este caso AuthWrapper envuelto en MultiProvider).**