Desarrollo de Interfaces

# Unidad 02. Introducción a Flutter y Dart




Autor: Sergi García


Actualizado Julio 2025

## Licencia



## Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

## ÍNDICE

# UNIDAD 02. INTRODUCCIÓN A FLUTTER Y A DART

## 1. INTRODUCCIÓN A FLUTTER Y DART

### 🔷 ¿Qué es Flutter?

Flutter es un framework de desarrollo de UI creado por Google para construir aplicaciones nativas compiladas para móvil, web y escritorio desde una única base de código. Fue lanzado oficialmente en diciembre de 2018 y su lenguaje de programación principal es Dart, también desarrollado por Google.

Características clave:

- UI declarativa: La interfaz se describe en función del estado actual de la app.
- Renderizado propio: Usa su propio motor gráfico (Skia), no depende de los componentes nativos del sistema operativo.
- Alto rendimiento: Compila a código nativo para ARM, x86 y web.
- Recarga en caliente (Hot Reload): Permite ver los cambios al instante sin reiniciar la aplicación.
- Amplia personalización de UI: Widgets totalmente personalizables y adaptables.

### 🔷 Ventajas del desarrollo multiplataforma con Flutter

1. Código único para múltiples plataformas: Escribes una sola vez, ejecutas en Android, iOS, Web y Desktop.
2. Desarrollo más rápido: Gracias a Hot Reload y al ecosistema bien integrado.
3. UI consistente: Al usar su propio motor de renderizado, no depende de las diferencias entre plataformas.
4. Comunidad activa y recursos: En constante crecimiento, con miles de paquetes disponibles.
5. Excelente documentación oficial: Con guías prácticas y detalladas.

### 🔷 Flutter vs React Native vs Apps Nativas

| Característica | Flutter | React Native | Nativo (Kotlin/Swift) |
|---|---|---|---|
| **Lenguaje** | Dart | JavaScript | Kotlin / Swift |
| **Rendimiento** | Alto (compilación nativa) | Medio-Alto | Excelente |
| **UI** | 100% personalizada con widgets | Bridged con componentes nativos | Componentes nativos |
| **Hot Reload** | Sí | Sí | No |
| **Comunidad y soporte** | Alta y creciendo | Muy grande | Alta pero separada por plataforma |
| **Acceso a funciones nativas** | Completo con plugins y canales | Requiere puente con código nativo | Directo |
| **Estabilidad** | Alta | Media-Alta | Alta |

## 2. Instalación de Flutter y Configuración del Entorno

### 🔧 Requisitos generales

Los requisitos para instalar Flutter son:

- Un sistema operativo compatible: Windows, macOS o Linux
- Espacio en disco: Al menos 2.8 GB (sin contar dependencias)
- Un editor de texto o IDE: Visual Studio Code, Android Studio, etc.
- Git instalado y accesible desde la terminal

Para instalar Flutter, sigue los pasos actualizados en https://docs.flutter.dev/get-started/install

## 3. Introducción a Dart

Dart es un lenguaje de programación desarrollado por Google. Es orientado a objetos, fuertemente tipado, con sintaxis similar a JavaScript/Java y pensado para la construcción de interfaces de usuario reactivas, como en Flutter.

### 🔷 Tipado en Dart

Dart es estáticamente tipado, pero puede inferir el tipo automáticamente.

```
int edad = 30;
double precio = 12.5;
bool activo = true;
String nombre = "Juan";

// Inferencia automática
var ciudad = "Madrid";     // String
final pais = "España";     // Constante en tiempo de ejecución
const pi = 3.1416;         // Constante en tiempo de compilación
```

- final: se asigna una sola vez, pero en tiempo de ejecución.
- const: se conoce su valor en tiempo de compilación.

### 🔷 Variables

```
var nombre = "Carlos";     // Inferido como String
String saludo = "Hola";
dynamic valor = 45;        // Puede cambiar de tipo (no recomendado salvo casos especiales)
valor = "Texto";
```

### 🔷 Funciones

```
// Función simple
String saludar(String nombre) {
  return "Hola, $nombre";
}
```

```dart
// Función flecha (arrow function)
int sumar(int a, int b) => a + b;

// Función con parámetros opcionales
void mostrarMensaje(String mensaje, [int veces = 1]) {
  for (int i = 0; i < veces; i++) {
    print(mensaje);
  }
}

// Parámetros con nombre
void crearUsuario({required String nombre, int edad = 18}) {
  print("Usuario: $nombre, Edad: $edad");
}
```

### 🔷 Clases en Dart

```dart
class Persona {
  String nombre;
  int edad;

  // Constructor
  Persona(this.nombre, this.edad);

  // Método
  void saludar() {
    print("Hola, soy $nombre y tengo $edad años");
  }
}

void main() {
  var persona = Persona("Lucía", 25);
  persona.saludar();
}
```

### 🔷 Herencia y sobrescritura

```dart
class Empleado extends Persona {
  String cargo;

  Empleado(String nombre, int edad, this.cargo) : super(nombre, edad);

  @override
```

```
  void saludar() {
    print("Hola, soy $nombre, trabajo como $cargo");
  }
}
```

🔷 **Estructuras útiles**

- Listas

```
List<String> frutas = ["Manzana", "Banana", "Pera"];
frutas.add("Uva");
```

- Mapas (diccionarios)

```
Map<String, dynamic> persona = {
  'nombre': 'Luis',
  'edad': 30,
};
```

- Conjuntos (Set)

```
Set<int> numeros = {1, 2, 3, 3};
```

## 4. ESTRUCTURA BÁSICA DE UNA APP FLUTTER Y WIDGETS FUNDAMENTALES

🔷 **Estructura básica de una aplicación Flutter**

Toda app Flutter comienza en el archivo main.dart dentro del directorio lib/. Este es el punto de entrada:

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mi primera app Flutter',
      home: HomePage(),
    );
  }
}
```

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Inicio')),
      body: Center(child: Text('Hola Mundo')),
    );
  }
}
```

🔷 **Explicación del código**

- main() → función principal que lanza la app con runApp().
- MyApp → widget raíz que define el diseño global.
- MaterialApp → proporciona navegación, temas, rutas, etc.
- Scaffold → estructura visual estándar con AppBar, Body, Drawer, etc.
- HomePage → pantalla principal.

🔷 **Tipos de widgets**

En Flutter todo es un widget, desde la estructura hasta el estilo. Existen dos tipos principales:

1. StatelessWidget
    - No guarda estado interno.
    - Redibujado solo si cambia el padre.

```
class MiWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('Soy un widget sin estado');
  }
}
```

2. StatefulWidget
    - Tiene un estado mutable.
    - Usa setState para redibujar la UI.

```
class Contador extends StatefulWidget {
  @override
  _ContadorState createState() => _ContadorState();
}

class _ContadorState extends State<Contador> {
  int contador = 0;

  void incrementar() {
    setState(() {
      contador++;
```

```
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Contador: $contador'),
        ElevatedButton(
          onPressed: incrementar,
          child: Text('Incrementar'),
        ),
      ],
    );
  }
}
```

🔷 **Widgets fundamentales**

| Widget | Descripción |
|---|---|
| Text | Muestra texto |
| Row / Column | Organiza widgets en horizontal/vertical |
| Container | Caja con padding, margen, color, etc. |
| Image | Muestra imágenes |
| ElevatedButton | Botón elevado con estilo Material |
| ListView | Lista desplazable |
| Stack | Superpone widgets |
| Expanded | Expande un hijo dentro de Row/Column |

## 5. WIDGETS DE DISPOSICIÓN (LAYOUT) EN FLUTTER

Los widgets de disposición controlan cómo se alinean, organizan y muestran los elementos en la pantalla. Son fundamentales para construir interfaces visuales responsivas y ordenadas.

🔷 **Row y Column**

- Row: organiza widgets horizontalmente
- Column: organiza widgets verticalmente

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
```

```
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Text("Elemento 1"),
    Text("Elemento 2"),
  ],
)
```

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: [
    Icon(Icons.home),
    Icon(Icons.star),
    Icon(Icons.settings),
  ],
)
```

Alineaciones comunes:

- MainAxisAlignment (eje principal):
  - start, center, end, spaceBetween, spaceAround, spaceEvenly
- CrossAxisAlignment (eje cruzado):
  - start, center, end, stretch

### 🔷 Container

Un widget de caja versátil:

```
Container(
  padding: EdgeInsets.all(16),
  margin: EdgeInsets.symmetric(horizontal: 10),
  color: Colors.blue,
  child: Text("Soy un Container"),
)
```

Propiedades clave:

- padding / margin
- width, height
- decoration: para bordes, sombras, bordes redondeados

### 🔷 Expanded y Flexible

Permiten que los widgets ocupen el espacio disponible dentro de un Row o Column.

```
Row(
  children: [
    Expanded(child: Container(color: Colors.red, height: 100)),
    Expanded(child: Container(color: Colors.green, height: 100)),
  ],
)
```

- Expanded: ocupa todo el espacio libre disponible
- Flexible: similar, pero con más control (puede ajustar a contenido si fit: FlexFit.loose)

### 🔷 Stack

Permite superponer widgets unos encima de otros.

```
Stack(
  children: [
    Container(width: 200, height: 200, color: Colors.blue),
    Positioned(
      top: 20,
      left: 20,
      child: Text("Encima"),
    ),
  ],
)
```

- Se usa mucho para overlays, banners, y composiciones avanzadas.

### 🔷 Padding y Align

```
Padding(
  padding: EdgeInsets.all(20),
  child: Text("Con espacio alrededor"),
)
```

```
Align(
  alignment: Alignment.centerRight,
  child: Text("Alineado a la derecha"),
)
```

### 🔷 SizedBox

Para espacios vacíos o tamaño fijo:

```
SizedBox(height: 20), // espacio vertical
SizedBox(width: 100, height: 100), // caja vacía de tamaño fijo
```

## 6. NAVEGACIÓN ENTRE PANTALLAS EN FLUTTER

### 🔷 ¿Qué es la navegación?

En Flutter, la navegación se refiere a moverse entre pantallas o vistas. Estas pantallas se llaman rutas (Route) y son gestionadas por un navegador (Navigator).

Flutter maneja una pila de rutas, similar a cómo funcionan los navegadores web: puedes "empujar" (push) una pantalla y "quitar" (pop) para volver.

### 🔷 Navegación básica usando Navigator.push y Navigator.pop

```
// Página principal
Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => SegundaPagina()),
);

// Volver atrás
Navigator.pop(context);

class SegundaPagina extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Segunda Página")),
      body: Center(
        child: ElevatedButton(
          onPressed: () => Navigator.pop(context),
          child: Text("Volver"),
        ),
      ),
    );
  }
}
```

### 🔷 Pasar datos entre pantallas

```
// Enviar datos
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => DetalleProducto(nombre: "Laptop", precio:
1299),
  ),
);

dart
CopiarEditar
// Recibir datos
class DetalleProducto extends StatelessWidget {
  final String nombre;
  final double precio;
```

```
  DetalleProducto({required this.nombre, required this.precio});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Detalle")),
      body: Text("Producto: $nombre - \$${precio.toStringAsFixed(2)}"),
    );
  }
}
```

### 🔷 Recibir un valor al volver

```
// Navegar y esperar un resultado
final resultado = await Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => SeleccionColor()),
);

print("Color seleccionado: $resultado");

dart
CopiarEditar
// En la segunda pantalla
Navigator.pop(context, "Rojo");
```

### 🔷 Rutas nombradas (Named Routes)

```
// main.dart
void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (context) => PantallaInicio(),
      '/perfil': (context) => PantallaPerfil(),
    },
  ));
}
```

```
// Navegar
Navigator.pushNamed(context, '/perfil');
```

```
// Volver
Navigator.pop(context);
```

## 7. GESTIÓN DE ESTADO EN FLUTTER

### 🔷 ¿Qué es el estado?

El estado es cualquier dato que puede cambiar durante la ejecución de la app y que afecta la interfaz. Ejemplos: el contador, el usuario autenticado, los datos de una lista, etc.

Flutter no impone un único patrón de gestión de estado. Vamos a ver varios enfoques, desde el más simple (setState) hasta los más escalables (Provider, Riverpod, Bloc).

#### 🔹 1. setState() (Estado local)

Ideal para apps pequeñas o cuando el cambio de estado solo afecta un widget.

```
class Contador extends StatefulWidget {
  @override
  _ContadorState createState() => _ContadorState();
}

class _ContadorState extends State<Contador> {
  int valor = 0;

  void incrementar() {
    setState(() {
      valor++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Valor: $valor"),
        ElevatedButton(
          onPressed: incrementar,
          child: Text("Incrementar"),
        ),
      ],
    );
  }
}
```

#### 🔹 2. Provider (Gestión de estado global)

Instalación:

```
dependencies:
  provider: ^6.1.1
```

Definir modelo de estado:

```
class ContadorModel extends ChangeNotifier {
  int _valor = 0;
  int get valor => _valor;

  void incrementar() {
    _valor++;
    notifyListeners();
  }
}
```

Integración:

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => ContadorModel(),
      child: MyApp(),
    ),
  );
}
```

Consumo en widgets:

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final contador = Provider.of<ContadorModel>(context);

    return Scaffold(
      appBar: AppBar(title: Text("Provider Demo")),
      body: Center(child: Text("Valor: ${contador.valor}")),
      floatingActionButton: FloatingActionButton(
        onPressed: contador.incrementar,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- 3. Riverpod (más moderno, escalable y desacoplado)

Instalación:

```
dependencies:
  flutter_riverpod: ^2.5.1
```

Ejemplo con StateNotifierProvider:

```
final contadorProvider = StateNotifierProvider<Contador, int>((ref) {
  return Contador();
});

class Contador extends StateNotifier<int> {
  Contador() : super(0);
  void incrementar() => state++;
}
```

Uso:

```
class Home extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final valor = ref.watch(contadorProvider);

    return Scaffold(
      body: Center(child: Text("Contador: $valor")),
      floatingActionButton: FloatingActionButton(
        onPressed: () =>
ref.read(contadorProvider.notifier).incrementar(),
        child: Icon(Icons.add),
      ),
    );
  }
}
```

🔹 **4. Bloc (Business Logic Component)**

Ideal para proyectos grandes con separación de lógica, UI y eventos.

Se basa en:

- Eventos (acciones del usuario)
- Estados (respuestas al cambio)
- Streams para manejar los cambios

## 8. CONSUMO DE APIS REST EN FLUTTER CON HTTP Y MODELOS EN DART

🔷 Dependencia necesaria

Agrega el paquete http a tu pubspec.yaml:

```
dependencies:
  http: ^0.13.6
```

### 🔷 Hacer una petición GET

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;

Future<void> obtenerDatos() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
  final respuesta = await http.get(url);

  if (respuesta.statusCode == 200) {
    final datos = jsonDecode(respuesta.body);
    print(datos);
  } else {
    throw Exception('Error al cargar datos');
  }
}
```

### 🔷 Crear un modelo en Dart

Supongamos que recibimos una lista de posts con esta estructura:

```
{
  "userId": 1,
  "id": 1,
  "title": "Título",
  "body": "Contenido del post"
}
```

Creamos un modelo:

```dart
class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

  Post({required this.userId, required this.id, required this.title,
required this.body});

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      userId: json['userId'],
```

```
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

### 🔷 Convertir respuesta en lista de objetos

```
Future<List<Post>> fetchPosts() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
  final response = await http.get(url);

  if (response.statusCode == 200) {
    final List<dynamic> lista = jsonDecode(response.body);
    return lista.map((json) => Post.fromJson(json)).toList();
  } else {
    throw Exception('Error al obtener posts');
  }
}
```

### 🔷 Mostrar datos en un ListView

```
class PostPage extends StatefulWidget {
  @override
  _PostPageState createState() => _PostPageState();
}

class _PostPageState extends State<PostPage> {
  late Future<List<Post>> _futurePosts;

  @override
  void initState() {
    super.initState();
    _futurePosts = fetchPosts();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Posts")),
      body: FutureBuilder<List<Post>>(
        future: _futurePosts,
        builder: (context, snapshot) {
```

```
          if (snapshot.connectionState == ConnectionState.waiting)
            return Center(child: CircularProgressIndicator());

          if (snapshot.hasError)
            return Center(child: Text("Error: ${snapshot.error}"));

          final posts = snapshot.data!;
          return ListView.builder(
            itemCount: posts.length,
            itemBuilder: (context, index) {
              final post = posts[index];
              return ListTile(
                title: Text(post.title),
                subtitle: Text(post.body),
              );
            },
          );
        },
      ),
    );
  }
}
```