

Desarrollo de Interfaces

# Unidad 02. Introducción a Flutter y Dart

---



Autor: Sergi García



Actualizado Julio 2025

## Licencia



**Reconocimiento - No comercial - CompartirIgual (BY-NC-SA):** No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se ha de hacer con una licencia igual a la que regula la obra original.

## 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

<b>1. Introducción a Flutter y Dart</b>	<b>3</b>
<b>2. Instalación de Flutter y Configuración del Entorno</b>	<b>4</b>
<b>3. Introducción a Dart</b>	<b>4</b>
<b>4. Estructura básica de una app Flutter</b>	<b>6</b>
<b>5. Widgets en Flutter</b>	<b>9</b>
<b>6. Resumen de principales Widgets</b>	<b>13</b>
<b>7. Widgets de Disposición (Layout) en Flutter</b>	<b>17</b>
<b>8. Navegación entre pantallas en Flutter</b>	<b>19</b>
<b>9. Gestión de Estado en Flutter</b>	<b>21</b>
<b>10. Consumo de APIs REST en Flutter con http y modelos en Dart</b>	<b>24</b>
<b>11. Diseño responsivo y adaptabilidad en Flutter</b>	<b>26</b>
<b>12. Recursos recomendados para aprender Flutter y Dart</b>	<b>29</b>
<b>13. Despliegue de Flutter en diferentes plataformas</b>	<b>31</b>

## UNIDAD 02. INTRODUCCIÓN A FLUTTER Y A DART

### 1. INTRODUCCIÓN A FLUTTER Y DART

#### ◆ ¿Qué es Flutter?

Flutter es un framework de código abierto desarrollado por Google para crear interfaces nativas multiplataforma (móvil, web y escritorio) desde una única base de código. Su primera versión estable se lanzó en diciembre de 2018, y desde entonces ha ganado popularidad por su rendimiento, flexibilidad y productividad.

##### **Características clave:**

- ✓ UI declarativa: La interfaz se construye en función del estado actual de la aplicación.
- ✓ Motor de renderizado propio (Skia): No depende de componentes nativos del sistema, lo que garantiza consistencia visual en todas las plataformas.
- ✓ Alto rendimiento: Compila a código nativo (ARM, x64) y JavaScript (para web).
- ✓ Hot Reload: Permite ver cambios al instante sin reiniciar la app, acelerando el desarrollo.
- ✓ Widgets altamente personalizables: Ofrece una amplia biblioteca de componentes adaptables.

#### ◆ ¿Qué es Dart y por qué Flutter lo usa?

Dart es el lenguaje de programación detrás de Flutter, diseñado por Google para ser:

- ✓ Rápido: Compilación AOT (ahead-of-time) para producción y JIT (just-in-time) para desarrollo ágil.
- ✓ Productivo: Sintaxis clara y moderna, similar a JavaScript/TypeScript, Java y C#.
- ✓ Orientado a UI: Ideal para aplicaciones reactivas gracias a su manejo eficiente de estados y eventos.

¿Por qué Flutter eligió Dart?

- Rendimiento cercano al nativo (evita el "puente JavaScript" de otros frameworks).
- Capacidad de compilación multiplataforma (móvil, web y desktop).
- Hot Reload nativo, algo difícil de lograr con otros lenguajes.

#### ◆ Ventajas del desarrollo con Flutter

- ✓ Código único para múltiples plataformas:
  - Escribe una vez y despliega en Android, iOS, web, Windows, macOS y Linux.
- ✓ Productividad elevada:
  - Hot Reload acelera las iteraciones de desarrollo.
  - Amplio ecosistema de paquetes (pub.dev) y herramientas integradas (Dart DevTools).
- ✓ UI consistente y personalizable:
  - Los widgets de Flutter se ven y funcionan igual en todas las plataformas, sin inconsistencias entre Android/iOS.
- ✓ Comunidad activa y soporte de Google:
  - Más de 150,000 paquetes disponibles en pub.dev.
  - Documentación oficial detallada y actualizada.
- ✓ Rendimiento competitivo:
  - Supera a soluciones basadas en JavaScript (como React Native) al evitar el "puente" entre lenguajes.

### ◆ 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";
}

// 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

```
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

```
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

### ◆ 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.

### A continuación damos una explicación más detallada de como es una App de Flutter para principiantes:

#### 🏠 Estructura básica de una app Flutter

Imagina que construir una app es como armar una casa. Necesitas planos (código) y materiales (widgets). Todo comienza en el archivo `main.dart` (la puerta de entrada!).

#### 📦 Partes principales del código

```
import 'package:flutter/material.dart'; // 📦 Traemos las
"herramientas" de Flutter
```

👉 Esto es como: Abrir tu caja de herramientas antes de construir.

**material.dart** contiene todos los widgets básicos (botones, textos, diseños).

#### 🚀 Función `main()` - El motor de la app

```
void main() {
  runApp(MyApp()); // 🚩 Inicia la app con el widget MyApp
}
```

👉 Así funciona:

- **`main()`** es como el interruptor de luz de tu casa (lo primero que se ejecuta).
- **`runApp()`** "enciende" la aplicación usando `MyApp` (el widget principal).

## MyApp - La raíz de todo

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp( //  Define el "estilo" de la app
      title: 'Mi primera app', // Nombre (para el sistema)
      home: HomePage(), //  Primera pantalla al abrir
    );
  }
}
```

### Claves:

- MaterialApp es el "diseñador de interiores":
  - Configura temas, rutas y la pantalla inicial (home).
- Es Stateless porque no cambia después de crearse.

## HomePage - La pantalla principal

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold( //  Estructura básica de pantalla
      appBar: AppBar(title: Text('Inicio')), //  Barra superior
      body: Center(child: Text('Hola Mundo')), //  Contenido central
    );
  }
}
```

### Partes del Scaffold (andamiaje):

Widget	Función	Ejemplo real
AppBar	Barra superior con título	Como el nombre en WhatsApp
Body	Área principal de la pantalla	El chat en WhatsApp
(Opcionales) Drawer, FloatingActionButton	Menú lateral o botón flotante	El menú de Gmail

### Consejos para recordar:

1. main.dart siempre es el punto de entrada.
2. MaterialApp envuelve toda la app (como un contenedor gigante).
3. Scaffold da estructura a cada pantalla (como paredes y techo).
4. Los widgets se anidan como muñecas rusas:  
MaterialApp > Scaffold > Center > Text.

### Ejemplo visual:



```

MaterialApp( // 🌐 La app completa
  home: Scaffold( // 🏠 Una pantalla
    body: Center( // 🔄 Centra contenido
      child: Text('Hola'), // 📝 Widget final
    ),
  ),
)

```

## 5. WIDGETS EN FLUTTER



### ¿Qué es un Widget en Flutter?

En Flutter, todo es un widget. Pero, ¿qué significa esto exactamente? Imagina que estás construyendo una casa de LEGO:

- **Widget = Pieza de LEGO** 🧱

Cada elemento visual o funcional en tu app (botones, textos, imágenes, pantallas, incluso la estructura completa) es un widget. Se combinan como bloques para crear interfaces complejas.



### Definición Técnica

Un widget es:

1. Componente reutilizable: Pequeña unidad de UI (Interfaz de Usuario) o lógica.
2. Configurable: Recibe parámetros (props) para personalizar su comportamiento/apariencia.
3. Anidable: Los widgets se componen unos dentro de otros (como un árbol).

Ejemplo visual:

```

Column( // ← Widget padre
  children: [
    Text('Hola'), // ← Widget hijo
    Icon(Icons.star), // ← Widget hijo
  ],
)

```



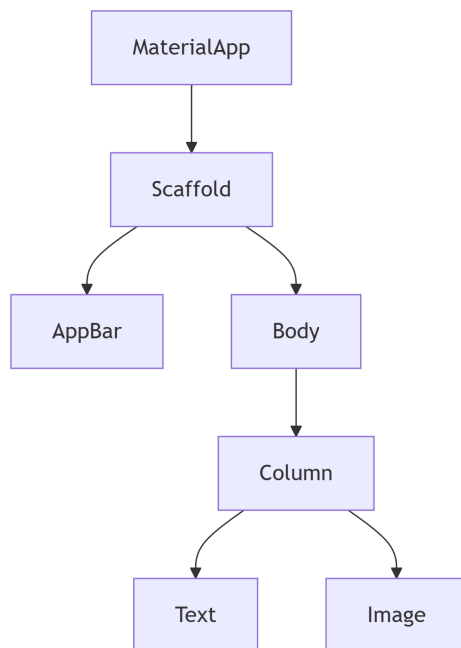
### Tipos de Widgets

Tipo	Ejemplos comunes	¿Para qué sirve?
Layout	Row, Column, Stack	Organizar otros widgets en pantalla.
Visual	Text, Image, Icon	Mostrar contenido estático.
Interactivo	ElevatedButton, TextField	Responder a acciones del usuario.
Estructural	Scaffold, Container	Proporcionar "esqueleto" a la UI.



### El Árbol de Widgets

Flutter organiza los widgets en una jerarquía de padres e hijos:



### ¿Por qué importa?

- Eficiencia: Flutter solo redibuja los widgets que cambian (¡no toda la pantalla!).
- Legibilidad: El código sigue la estructura visual.

### Ejemplo Práctico: Widget Personalizado

Crear un widget `BotonPersonalizado` (Stateless):

```
class BotonPersonalizado extends StatelessWidget {
  final String texto;
  final Color color;

  BotonPersonalizado(this.texto, this.color); // ← Parámetros

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(12),
      color: color,
      child: Text(texto),
    );
  }
}
```

// Uso:

```
BotonPersonalizado('Aceptar', Colors.green),
```

### ¿Por qué Flutter usa widgets?

1. Modularidad: Puedes reusar widgets en cualquier parte.
2. Declarativo: Describe QUÉ quieres mostrar (no CÓMO hacerlo paso a paso).
3. Optimización: Flutter gestiona automáticamente su renderizado.

### Regla de Oro

"En Flutter, si lo ves en pantalla, es un widget. Si no se ve, pero afecta a otros (como gestores de estado), también puede ser un widget."

### Widgets con y sin Estado

En Flutter, la interfaz de usuario se construye mediante widgets, que pueden clasificarse según su capacidad para manejar estado (datos que pueden cambiar durante la ejecución). Esta distinción es clave para el rendimiento y la arquitectura de la app.

### Definición Formal

#### 1. StatelessWidget



- **Inmutables:** No almacenan datos modificables después de su creación.
- **Renderizado:** Solo se redibujan cuando reciben nuevos parámetros (via props del padre).
- **Ciclo de vida:** Simplemente, construyen su UI una vez (build()).

#### 2. StatefulWidget

- **Mutable:** Gestionan un estado interno (State) que puede cambiar dinámicamente.
- **Renderizado:** Actualizan la UI cuando se llama setState().
- **Ciclo de vida:** Complejo (initState, dispose, etc.) para manejar recursos.

### Analogía Sencilla (Stateless vs Stateful)

Imagina una caja de herramientas:

- **Stateless**  Como un martillo.
  - Siempre hace lo mismo: clavar.
  - No "recuerda" nada después de usarlo.
- **Stateful**  Como un termómetro digital.
  - Mide y actualiza la temperatura (estado).
  - Reacciona a cambios externos.

### 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'),  
        ),  
      ],  
    );  
  }  
}
```

## 6. RESUMEN DE PRINCIPALES WIDGETS



### Widgets Fundamentales en Flutter

Estos son los "ladrillos esenciales" para construir interfaces en Flutter. Cada uno resuelve necesidades específicas de diseño y funcionalidad.



#### Text

Descripción: Muestra texto en pantalla con estilo.

Propiedades clave:

- style: Fuente, tamaño, color (usar TextStyle).
- textAlign: Alineación (centro, izquierda, etc.).

Ejemplo:

```
Text(  
  'Hola Flutter!',  
  style: TextStyle(  
    fontSize: 24,  
    fontWeight: FontWeight.bold,  
    color: Colors.blue,  
  ),  
)
```



#### Row / Column

Descripción:

- Row: Organiza widgets horizontalmente.
- Column: Organiza widgets verticalmente.

Propiedades clave:

- mainAxisAlignment: Alineación en el eje principal (ej: MainAxisAlignment.center).
- crossAxisAlignment: Alineación en el eje secundario.
- children: Lista de widgets hijos.

Ejemplo comparativo:

```
Row( // ←→  
  children: [  
    Icon(Icons.star),  
    Text('Fila'),  
  ],  
)
```

```
Column( // ⇅  
  children: [  
    Icon(Icons.star),  
    Text('Columna'),  
  ],  
)
```

## Container

Descripción: "Caja" personalizable con decoración y espaciado.

Propiedades clave:

- padding: Espacio interno.
- margin: Espacio externo.
- decoration: Color, bordes, sombras (BoxDecoration).
- width/height: Tamaño fijo.

Ejemplo:

```
Container(  
  padding: EdgeInsets.all(16),  
  margin: EdgeInsets.symmetric(vertical: 8),  
  decoration: BoxDecoration(  
    color: Colors.amber,  
    borderRadius: BorderRadius.circular(10),  
  ),  
  child: Text('Contenedor'),  
)
```

## Image

Descripción: Muestra imágenes desde diferentes fuentes.

Tipos de carga:

- Image.asset('ruta/local'): Desde archivos del proyecto.
- Image.network('URL'): Desde internet.

Ejemplo:

```
Image.network(  
  'https://example.com/imagen.jpg',  
  width: 200,  
  fit: BoxFit.cover, // Ajuste de la imagen  
)
```

## ElevatedButton

Descripción: Botón con elevación visual (Material Design).

Propiedades clave:

- onPressed: Función al presionar (si es null, se deshabilita).
- child: Widget hijo (texto, icono, etc.).

Ejemplo:

```
ElevatedButton(  
  onPressed: () {  
    print('Botón presionado!');  
  },  
  child: Text('Presiona aquí'),  
)
```

## **ListView**

Descripción: Lista desplazable (vertical u horizontal).

Casos de uso:

**Lista básica:**

```
ListView(  
  children: [  
    ListTile(title: Text('Item 1')),  
    ListTile(title: Text('Item 2')),  
  ],  
)
```

**Dinámica (para muchos elementos):**

```
ListView.builder(  
  itemCount: 100,  
  itemBuilder: (context, index) {  
    return ListTile(title: Text('Item $index'));  
  },  
)
```

## **Stack**

Descripción: Superpone widgets (útil para elementos en capas).

**Ejemplo común:** Texto sobre imagen.

```
Stack(  
  children: [  
    Image.network('https://example.com/fondo.jpg'),  
    Positioned( // Posiciona un hijo relativo al Stack  
      bottom: 10,  
      child: Text('Texto superpuesto'),  
    ),  
  ],  
)
```

## **Expanded**

Descripción: Ocupa el espacio disponible en Row/Column.

Regla clave: Solo funciona dentro de Row o Column.

**Ejemplo:**

```
Row(  
  children: [  
    Expanded( // ← Ocupa 70% del espacio  
      flex: 7,  
      child: Container(color: Colors.red),  
    ),  
    Expanded( // ← Ocupa 30%  
      flex: 3,  
    ),  
  ],  
)
```

```

        child: Container(color: Colors.blue),
      ),
    ],
  ),
)

```

### Resumen de casos de uso

Widget	¿Cuándo usarlo?	Ejemplo real
Text	Mostrar títulos, descripciones.	Nombre de usuario.
Row/Column	Formularios, barras de herramientas.	Fila de iconos en redes.
Container	Agrupar widgets con estilo común.	Tarjeta con sombra.
ListView	Listas largas (mensajes, productos).	Chat de WhatsApp.
Stack	Botones flotantes, imágenes con texto.	Foto de perfil con badge.

### Ejemplo Integrado

Combina varios widgets para crear una tarjeta de producto:

```

Container(
  margin: EdgeInsets.all(10),
  decoration: BoxDecoration(
    border: Border.all(color: Colors.grey),
  ),
  child: Column(
    children: [
      Image.network('https://example.com/producto.jpg'),
      Padding(
        padding: EdgeInsets.all(8),
        child: Row(
          children: [
            Expanded(
              child: Text('Zapatos deportivos'),
            ),
            ElevatedButton(
              onPressed: () {},
              child: Text('Comprar'),
            ),
          ],
        ),
      ),
    ],
  ),
)

```



```

    ),
  ),
],
),
)

```

## 7. 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
```

## 8. 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),
),
);
```

```
// 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");
```

```
// 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);

```

## 9. 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

## 10. 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

```
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:

```
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),
              );
            },
          );
        },
      ),
    );
  }
}
```

## 11. DISEÑO RESPONSIVO Y ADAPTABILIDAD EN FLUTTER

En el desarrollo móvil, las aplicaciones deben verse y funcionar correctamente en diferentes tamaños de pantalla (móviles, tablets e incluso web). Flutter ofrece herramientas poderosas para crear interfaces que se adapten automáticamente.

## Conceptos Clave

1. **MediaQuery** → Obtiene información del dispositivo (tamaño de pantalla, orientación, etc.).

```
final screenSize = MediaQuery.of(context).size;
final screenWidth = screenSize.width;
final screenHeight = screenSize.height;
final isPortrait = MediaQuery.of(context).orientation ==
Orientation.portrait;
```

2. **LayoutBuilder** → Similar a MediaQuery, pero se ajusta dinámicamente cuando cambia el espacio disponible.

```
LayoutBuilder(
  builder: (context, constraints) {
    if (constraints.maxWidth > 600) {
      return TabletLayout(); // Diseño para tablets
    } else {
      return MobileLayout(); // Diseño para móviles
    }
  },
)
```

3. **Expanded y Flexible** → Distribuyen espacio en Row/Column.
  - Expanded → Ocupa todo el espacio restante.
  - Flexible → Similar, pero con flexibilidad en su tamaño.

```
Row(
  children: [
    Expanded( // Ocupa 70%
      flex: 7,
      child: Container(color: Colors.red),
    ),
    Expanded( // Ocupa 30%
      flex: 3,
      child: Container(color: Colors.blue),
    ),
  ],
)
```

## Ejemplo Práctico: App Responsiva

Objetivo: Crear una app que muestre una lista en móvil (vertical) y en tablet (grid horizontal).

```
class ResponsiveLayout extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('App Responsiva')),
    );
  }
}
```

```

body: LayoutBuilder(
  builder: (context, constraints) {
    // Tablet (ancho mayor a 600px)
    if (constraints.maxWidth > 600) {
      return GridView.builder(
        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
          crossAxisCount: 2, // 2 columnas
        ),
        itemBuilder: (context, index) => ItemWidget(index),
      );
    }
    // Móvil (una columna)
    else {
      return ListView.builder(
        itemBuilder: (context, index) => ItemWidget(index),
      );
    }
  },
),
);
}
}

class ItemWidget extends StatelessWidget {
  final int index;
  ItemWidget(this.index);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Center(child: Text('Ítem $index')),
    );
  }
}

```



### Técnicas Avanzadas

#### 1. Breakpoints Personalizados

Define tamaños estándar para diferentes dispositivos:

```

bool isMobile(BuildContext context) =>
MediaQuery.of(context).size.width < 600;
bool isTablet(BuildContext context) =>
MediaQuery.of(context).size.width >= 600;

```

## 2. **OrientationBuilder** → Detecta cambios entre vertical/horizontal.

```
OrientationBuilder(
  builder: (context, orientation) {
    return orientation == Orientation.portrait
      ? PortraitLayout()
      : LandscapeLayout();
  },
)
```

## 3. **FractionallySizedBox** → Widget que ocupa un % del espacio disponible.

```
FractionallySizedBox(
  widthFactor: 0.8, // 80% del ancho disponible
  child: ElevatedButton(onPressed: () {}, child: Text('Botón')),
)
```

### ✓ Buenas Prácticas

- ✓ Evitar dimensiones fijas → Usar MediaQuery o porcentajes.
- ✓ Testear en múltiples dispositivos → Emuladores y dispositivos reales.
- ✓ Priorizar flexibilidad → Usar Expanded, Flexible, y Wrap.

### **Ejemplo Final: Adaptando una App Real**

Imagina una app de noticias:

- Móvil: Lista vertical con imagen + título.
- Tablet: Grid con imagen, título y resumen.
- Web: Barra lateral + contenido principal.

```
Widget build(BuildContext context) {
  final screenWidth = MediaQuery.of(context).size.width;




  if (screenWidth > 1200) {
    return WebLayout(); // Diseño para web
  } else if (screenWidth > 600) {
    return TabletLayout(); // Diseño para tablet
  } else {
    return MobileLayout(); // Diseño para móvil
  }
}
```

## 12. RECURSOS RECOMENDADOS PARA APRENDER FLUTTER Y DART


### **Documentación Oficial**

- Flutter Docs  
<https://docs.flutter.dev>
- Dart Language  
<https://dart.dev/language>



### Cursos Gratuitos

1. Flutter Crash Course (Google)  
 <https://docs.flutter.dev/get-started/codelab>
2. Dart en Codecademy  
 <https://www.codecademy.com/learn/learn-dart>
3. Curso Completo de Flutter (YouTube - Fernando Herrera)  
 [https://www.youtube.com/watch?v=GXIJJkq\\_H8g&list=PLV6pYUAZ-ZoE6kzN1t9IfV9aYkRFYhwyj](https://www.youtube.com/watch?v=GXIJJkq_H8g&list=PLV6pYUAZ-ZoE6kzN1t9IfV9aYkRFYhwyj)




### Paquetes y Librerías

- Pub.dev (Repositorio Oficial)  
 <https://pub.dev>
- Flutter Awesome (Inspiración UI)  
 <https://flutterawesome.com>





### Libros

1. "Flutter in Action" (Manning)  
 <https://www.manning.com/books/flutter-in-action>
2. "Dart Apprentice" (Ray Wenderlich)  
 <https://www.raywenderlich.com/books/dart-apprentice>



### Comunidad

- Stack Overflow (Flutter Tag)  
 <https://stackoverflow.com/questions/tagged/flutter>
- Reddit r/FlutterDev  
 <https://www.reddit.com/r/FlutterDev>
- Flutter Community en Medium  
 <https://medium.com/flutter-community>

### Herramientas Clave

1. Flutter DevTools (Debugging)  
 <https://docs.flutter.dev/tools/devtools>
2. Firebase para Flutter  
 <https://firebase.flutter.dev>
3. Riverpod (Gestión de Estado)  
 <https://riverpod.dev>
4. VS Code + Extensión Flutter  
 <https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter>

### Extra: Proyectos Open-Source

- Repositorio Oficial de Flutter  
 <https://github.com/flutter/flutter>
- Ejemplos de Apps en GitHub  
 <https://github.com/flutter/samples>

## 13. DESPLIEGUE DE FLUTTER EN DIFERENTES PLATAFORMAS



### Android



Documentación oficial para publicar en Google Play Store

<https://docs.flutter.dev/deployment/android>

- Cómo generar APK/AAB: flutter build appbundle o flutter build apk
- Configuración de firma: keytool y build.gradle
- Proceso de subida a Play Console



### iOS



Guía completa para App Store

<https://docs.flutter.dev/deployment/ios>

- Requisitos: Cuenta de desarrollador Apple (\$99/año)
- Generar IPA: flutter build ipa
- Configurar Xcode: Certificados y provisionamiento



### Windows



Compilación para Windows

<https://docs.flutter.dev/deployment/windows>

- Requisitos: Visual Studio 2022 con workloads específicos
- Comando: flutter build windows
- Opciones de empaquetado: MSIX, EXE o instalador



### Linux



Despliegue en Linux

<https://docs.flutter.dev/deployment/linux>

- Dependencias: GTK, CMake y clang
- Formatos soportados: .deb, .rpm y Snap
- Personalización: Íconos y metadatos



### Web



Publicación para web

<https://docs.flutter.dev/deployment/web>

- Optimización: flutter build web --web-renderer canvaskit
- Plataformas de hosting recomendadas:
  - Firebase: <https://firebase.google.com/docs/hosting>
  - GitHub Pages: gh-pages branch
  - Netlify/Vercel: Drag-and-drop deployment



### Pasos Comunes

1. Build general:
2. bash
3. flutter build <platform> # android, ios, windows, linux, web
4. Pruebas locales:
5. bash
6. flutter run -d <device> # chrome, edge, dispositivo físico
7. Requisitos previos:
  - Android: JDK 11+, Android Studio

- iOS: Xcode 14+, macOS
- Windows: Visual Studio 2022
- Linux: GTK 3.0+
- Web: Chrome para testing



### Consideraciones Especiales

- Android/iOS:
  - Necesitas cuentas de desarrollador
  - Proceso de revisión en tiendas (1-3 días)
- Windows/Linux:
  - No requieren tiendas oficiales
  - Puedes distribuir directamente los ejecutables
- Web:
  - Soporte para PWA (Service Workers)
  - Optimiza assets con --tree-shake-icons



### Recursos Adicionales

- Flutter DevTools (Análisis de performance):  
<https://docs.flutter.dev/tools/devtools>
- Firebase Hosting (Para web apps):  
<https://firebase.google.com/docs/hosting>
- App Store Connect Help:  
<https://help.apple.com/app-store-connect>



### Flujo Recomendado

1. Prueba en modo debug (flutter run)
2. Build en modo release (flutter build --release)
3. Verifica con DevTools
4. Sigue la guía específica de cada plataforma
5. Publica y monitorea crashes con Firebase Crashlytics