Desarrollo de Interfaces

Unidad 11 - Navegación, estado y responsive en Flutter

Autor: Sergi García

Actualizado Septiembre 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 11. NAVEGACIÓN, ESTADO Y RESPONSIVE EN FLUTTER

1. INTRODUCCIÓN

Flutter se ha convertido en uno de los frameworks más versátiles y potentes para el desarrollo multiplataforma, permitiendo crear aplicaciones para iOS, Android, web y escritorio a partir de un único código base. Su enfoque declarativo y su gran ecosistema de paquetes lo convierten en una herramienta ideal tanto para principiantes como para desarrolladores que buscan construir aplicaciones escalables y de nivel profesional.

Flutter no solo destaca por su velocidad y versatilidad al desarrollar para múltiples plataformas, sino también porque ofrece las bases para crear **aplicaciones robustas, escalables y con una experiencia de usuario profesional**. Sin embargo, para alcanzar ese nivel no basta con conocer los widgets básicos: es necesario dar un paso más y dominar conceptos de arquitectura, navegación, estado y diseño adaptable.

En este tema profundizaremos en esos pilares que marcan la diferencia entre una app de prueba y un producto real listo para producción. Veremos cómo estructurar mejor el código con **patrones de arquitectura como MVVM**, cómo construir una navegación clara y mantenible, y cómo elegir el **gestor de estado adecuado** según la complejidad del proyecto. También aprenderemos a **diseñar interfaces responsivas** que se adapten a móviles, tablets o incluso la web.

El objetivo es que, al finalizar, no solo sepas cómo crear pantallas en Flutter, sino que tengas una **visión integral de cómo organizar, escalar y mantener proyectos de manera profesional**, con prácticas y herramientas que te servirán en el día a día del desarrollo.

2. MVVM EN FLUTTER: EXPLICACIÓN TEÓRICA

📚 ¿Qué es MVVM?

MVVM (Model–View–ViewModel) es un patrón de arquitectura de software que divide una aplicación en tres capas bien diferenciadas:

- **Model** → Representa los datos y la lógica de negocio.
  - Ejemplo: una lista de tareas, la respuesta de una API, o el acceso a base de datos.
- **View** → Es la interfaz de usuario. Se encarga de mostrar la información y capturar las interacciones del usuario.
  - Ejemplo: una pantalla con la lista de tareas y un botón para añadir nuevas.
- **ViewModel** → Actúa como intermediario entre el Model y la View.
  - Solicita los datos al Model.
  - Los transforma en información lista para mostrar.
  - Gestiona el estado: qué mostrar en cada momento (cargando, error, datos disponibles).

En Flutter, esta separación es especialmente útil porque evita que toda la lógica quede mezclada dentro de un **único widget gigante**, lo que complica el mantenimiento.

🧩 Analogía Didáctica

Imagina un **restaurante**:

- **Model** → **la cocina**: donde están los ingredientes y se preparan los platos (los datos).
- **View** → **el comedor**: los clientes ven el menú y disfrutan la comida (la interfaz de usuario).
- **ViewModel** → **el camarero**: recibe el pedido del cliente, lo comunica a la cocina, y luego entrega el plato listo.

👉 El punto clave es que el cliente **nunca habla directamente con la cocina**. Del mismo modo, en Flutter la View nunca accede directamente a los datos, sino que se comunica siempre a través del ViewModel.

💡 ¿Por qué usar MVVM en Flutter?

Flutter permite crear interfaces muy rápido, pero si no organizamos el código, acabamos con:

- Widgets que mezclan **UI + lógica + estado** en el mismo archivo.
- Dificultad para **escalar** el proyecto.
- Código **poco reutilizable**.
- Problemas de **testabilidad**, ya que la lógica depende de la UI.

Con MVVM ganamos:

✅ Separación de responsabilidades

- La View se centra en mostrar datos.
- El ViewModel en manejar la lógica y el estado.
- El Model en gestionar los datos.

✅ Consistencia

- Todas las pantallas siguen una estructura clara y predecible.

✅ Código mantenible y flexible

- Cambiar la UI no rompe la lógica.
- Modificar una API no obliga a rehacer la interfaz.

En Flutter, el **ViewModel suele implementarse con herramientas de gestión de estado** como **Provider**, **Riverpod** o **Bloc**, que permiten que la UI reaccione automáticamente a los cambios de datos.

🎯 Ejemplos Didácticos en apps reales

- **App Lista de tareas**
    - **Model**: lista de tareas (con título, fecha, estado).
    - **ViewModel**: gestiona añadir, borrar, marcar como completada. Expone una lista "lista para mostrar".
    - **View**: solo muestra la lista y botones.

- **App Comercio electrónico**
  - **Model**: productos con precio, stock, imágenes.
  - **ViewModel**: gestiona carrito, calcula total, aplica descuentos.
  - **View**: catálogo de productos y botón de "comprar".

✅ **En Resumen**

- **MVVM** = Modelo + Vista + ViewModel.
- Aporta **orden y claridad**: cada parte tiene un rol definido.
- Permite apps más **mantenibles, escalables y testeables**.
- En Flutter, el ViewModel suele implementarse con herramientas como **Provider** o **Riverpod** para manejar el estado de forma reactiva.

👉 Explicado sencillo: **MVVM evita que la lógica y la interfaz se mezclen, como separar la cocina del comedor con un camarero en medio.**

3. NAVEGACIÓN ENTRE PANTALLAS EN FLUTTER

En cualquier aplicación real no basta con mostrar una única pantalla. Los usuarios esperan poder:

- Iniciar sesión y luego ir al **dashboard**.
- Abrir el **detalle de un producto** desde una lista.
- Rellenar un **formulario de registro** y volver a la página anterior.
- Moverse entre **secciones distintas** con menús o tabs.

A este proceso de ir de una pantalla a otra lo llamamos **navegación**.

En Flutter, cada pantalla o vista suele representarse como un Widget (normalmente un Scaffold), y el framework maneja un stack de rutas (pila de pantallas). Cuando el usuario abre una nueva pantalla, esta se apila encima de la actual; cuando retrocede, la pantalla superior se desapila y vuelve a la anterior.

👉 **En otras palabras:** la navegación en Flutter funciona de forma muy parecida a un navegador web o a un conjunto de tarjetas apiladas: pones una encima (nueva pantalla) y cuando la quitas vuelves a la anterior.

¿Por qué es importante la navegación?

- **Organiza la experiencia de usuario:** divide la aplicación en secciones claras en lugar de tener todo en una sola pantalla.
- **Permite reutilizar pantallas:** por ejemplo, un mismo formulario puede abrirse desde diferentes lugares.
- **Gestiona el flujo de la aplicación:** login → home → detalle → carrito → confirmación.
- **Soporta deep linking y web:** en Flutter web, la URL debe reflejar en qué parte de la app estás.

Sin un sistema de navegación claro, la aplicación se convierte en un único widget gigante difícil de usar y de mantener.

Aquí tienes una explicación práctica y en profundidad de las técnicas de navegación más comunes en Flutter: Navigator.push / Navigator.pop, paso de datos entre pantallas, rutas nombradas y la alternativa moderna GoRouter.

3.1. Navigator.push y Navigator.pop (imperativo, stack-based)

Concepto: Flutter mantiene una pila de rutas (screens). Navigator.push apila una nueva ruta; Navigator.pop la desapila y vuelve a la anterior. Es el modo imperativo tradicional y sencillo para navegación puntual. Documentación de Flutter

Ejemplo completo (push & pop + recibir resultado)

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Push / Pop Demo',
      theme: ThemeData(primarySwatch: Colors.indigo),
      home: const HomeScreen(),
    );
  }
}

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});
  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  String? _selection;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Home')),
      body: Center(
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          ElevatedButton(
            onPressed: () async {
              // Lanzamos la segunda pantalla y esperamos el resultado
(Future)
              final result = await Navigator.push<String>(
                context,
                MaterialPageRoute(builder: (_) => const SelectionScreen()),
              );
              // result será lo pasado por Navigator.pop(context, value)
              setState(() => _selection = result);
```

```
              if (result != null) {
                ScaffoldMessenger.of(context).showSnackBar(
                  SnackBar(content: Text('Resultado: $result')),
                );
              }
            },
            child: const Text('Ir a selección'),
          ),
          const SizedBox(height: 16),
          Text('Selección actual: ${_selection ?? "--"}'),
        ]),
      ),
    );
  }
}

class SelectionScreen extends StatelessWidget {
  const SelectionScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Seleccionar')),
      body: Center(
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          ElevatedButton(
            onPressed: () => Navigator.pop(context, 'Opción A'),
            child: const Text('Elegir Opción A'),
          ),
          ElevatedButton(
            onPressed: () => Navigator.pop(context, 'Opción B'),
            child: const Text('Elegir Opción B'),
          ),
          const SizedBox(height: 12),
          ElevatedButton(
            onPressed: () => Navigator.pop(context), // sin resultado
            child: const Text('Cancelar'),
          ),
        ]),
      ),
    );
  }
}
```

Notas prácticas

- Navigator.push devuelve un Future que completa cuando se hace pop(...) en la ruta nueva. Úsalo para esperar valores (por ejemplo: seleccionar un elemento). Documentación de Flutter

- Si necesitas reemplazar la ruta actual (por ejemplo, después de login), usa pushReplacement o pushAndRemoveUntil.

3.2. Paso de datos entre pantallas

Hay dos patrones comunes:

A) Pasar datos al empujar la ruta (constructor / MaterialPageRoute)

// ejemplo rápido (fragmento)

```
final item = Item(id: 42, title: 'Hola');
Navigator.push(
  context,
  MaterialPageRoute(builder: (ctx) => DetailScreen(item: item)),
);
```

En este enfoque la pantalla destino recibe los datos por su constructor (tipado y claro).

B) Rutas nombradas con arguments → extraer con ModalRoute.of(context) o onGenerateRoute

- Puedes pasar un objeto a Navigator.pushNamed(..., arguments: yourObject) y extraerlo en la ruta destino. Es útil para navegación desde diferentes lugares sin importar la clase concreta de la pantalla.
- Más información en https://docs.flutter.dev/cookbook/navigation/navigate-with-arguments

Ejemplo completo con onGenerateRoute y argumentos

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

void main() => runApp(const MyApp());

class DetailArgs {
  final int id;
  final String title;
  DetailArgs(this.id, this.title);
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Named Routes with Args',
      initialRoute: '/',
      onGenerateRoute: (settings) {
        if (settings.name == DetailScreen.routeName) {
          final args = settings.arguments as DetailArgs;
          return MaterialPageRoute(
            builder: (_) => DetailScreen(id: args.id, title: args.title),
```

```
            settings: settings,
          );
        }
        // default route
        return MaterialPageRoute(builder: (_) => const HomeScreen());
      },
    );
  }
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) {
    final args = DetailArgs(99, 'Detalle desde Home');
    return Scaffold(
      appBar: AppBar(title: const Text('Home (named)')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, DetailScreen.routeName, arguments:
args);
          },
          child: const Text('Ir a detalle (named + args)'),
        ),
      ),
    );
  }
}

class DetailScreen extends StatelessWidget {
  static const routeName = '/detail';
  final int id;
  final String title;
  const DetailScreen({super.key, required this.id, required this.title});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Detalle #$id')),
      body: Center(child: Text('Título: $title')),
    );
  }
}
```

Consejo: Si tu app escala mucho, evita pasar objetos enormes por arguments en todas partes; para datos compartidos es más robusto usar un state manager (Provider, Riverpod, Bloc).

3.3. Rutas nombradas: cuándo y cómo

Flutter permite declarar rutas con **routes / onGenerateRoute** y navegar con **pushNamed**. Nota importante: la documentación oficial menciona que las rutas nombradas ya no son recomendadas para la mayoría de apps modernas cuando se requiere deep linking, web o un control fino del stack; en esos casos es preferible la API de Router o un paquete declarativo como go_router. No obstante, las rutas nombradas siguen siendo útiles para apps simples o migraciones. Más información en https://docs.flutter.dev/cookbook/navigation/navigate-with-arguments

Pros de rutas nombradas

- Centralización de rutas.
- Fácil de llamar desde cualquier parte (solo el nombre).

Contras

- Menos flexibles para deep links complejos y control del stack.
- Menos declarativas que Navigator 2.0 / GoRouter.

3.4. Go Router (declarativo, URL-friendly, recomendado para apps complejas / web)

Qué es: go_router es un paquete declarativo que usa la API de Router de Flutter para facilitar rutas basadas en URL, deep links, parámetros en path y query, redirecciones y navegación más predecible. Es la opción recomendada para apps con rutas complejas o targeting web + mobile.

Idea clave (go vs push):

- **context.push(...)** añade una página encima de la pila (similar a Navigator.push)
- **context.go(...)** navega a una ubicación declarativa; según la jerarquía de rutas puede reemplazar rutas intermedias (comportamiento más declarativo). Entender la diferencia es importante para predecir el estado del stack.

Ejemplo completo con go_router (path parameter + extra + MaterialApp.router)

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

void main() => runApp(const MyApp());

final GoRouter _router = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      name: 'home',
      builder: (context, state) => const HomeScreen(),
      routes: [
        // subruta con parametro en path
        GoRoute(
          path: 'details/:id',
          name: 'details',
          builder: (context, state) {
            final id = state.params['id']!; // path param
            final extra = state.extra as String?; // extra (opcional)
```

```
            return DetailsScreen(id: id, extra: extra);
          },
        ),
      ],
    ),
  ],
);

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp.router(
      title: 'GoRouter Demo',
      routerConfig: _router,
      debugShowCheckedModeBanner: false,
    );
  }
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Home (GoRouter)')),
      body: Center(
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          ElevatedButton(
            onPressed: () {
              // navegar usando path param
              context.go('/details/42');
            },
            child: const Text('Ir a details/42 (context.go)'),
          ),
          ElevatedButton(
            onPressed: () async {
              // push mantiene el stack y permite que al hacer pop volvamos a
esta pantalla
              await context.push('/details/99', extra: 'Extra desde Home');
              // Aquí podemos ejecutar lógica tras volver (si se usó push y
se hizo pop)
              ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(content: Text('Volvimos desde details
(push)')),
              );
            },
```

```
            child: const Text('Push /details/99 (context.push + extra)'),
          ),
        ]),
      ),
    );
  }
}

class DetailsScreen extends StatelessWidget {
  final String id;
  final String? extra;
  const DetailsScreen({super.key, required this.id, this.extra});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Details #$id')),
      body: Center(
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          Text('ID: $id'),
          if (extra != null) Text('Extra: $extra'),
          const SizedBox(height: 12),
          ElevatedButton(
            onPressed: () => context.pop(), // equivalente a
Navigator.pop(context)
            child: const Text('Volver'),
          ),
        ]),
      ),
    );
  }
}
```

Notas GoRouter

- Puedes usar state.params (path params), state.uri.queryParameters (query params) y state.extra (objetos) para pasar datos.
- context.goNamed('name', params: {...}, queryParams: {...}) es útil cuando defines name en GoRoute.
- push vs go: elige push si quieres apilar y luego volver al punto anterior; usa go cuando quieres navegar declarativamente a una ubicación (útil para deep links y rutas "salto directo"). Code With AndreaCodemagic blog

3.5. Buenas prácticas y cuándo usar cada técnica

- Navigator.push / MaterialPageRoute — ideal para demostraciones rápidas o pantallas modales/temporales. Fácil y directo.
- Rutas nombradas (pushNamed) — aceptable en apps simples; si tu app crece y necesita

deep linking, reconsidera.

- onGenerateRoute — útil para validar/centrar la creación de rutas con argumentos tipados.
- go_router (o Router API) — recomendado para apps complejas, web, deep linking, autenticación y redirecciones. Maneja parámetros, query y extra de forma declarativa.
- Evita pasar objetos gigantes por argumentos repetidamente; usa un state manager para datos compartidos (Provider, Riverpod, Bloc).
- Control del stack: aprende métodos como pushReplacement, pushAndRemoveUntil y las diferencias go/push en go_router.

4. GESTIÓN DE ESTADO EN FLUTTER

🔷 ¿Qué es el estado?

El estado en Flutter es cualquier dato que cambia con el tiempo y que afecta directamente a la interfaz de usuario.

Algunos ejemplos comunes:

- El valor de un contador.
- El usuario autenticado en la aplicación.
- Los items de un carrito de compras.

Cuando el estado cambia, Flutter necesita reconstruir los widgets que dependen de él, para reflejar la nueva información en pantalla.

🔷 ¿Por qué necesitamos gestores de estado?

En Flutter, cada vez que un dato cambia, es necesario reconstruir los widgets que dependen de él para que la interfaz muestre la información actualizada.

Si trabajamos con aplicaciones pequeñas (por ejemplo, un simple contador), podemos manejar estos cambios fácilmente con setState(). Sin embargo, a medida que la aplicación crece, aparecen varios problemas:

- **Complejidad en el manejo de datos**
  - Una aplicación real suele tener muchos datos cambiantes: usuario autenticado, productos de un carrito, temas de configuración, conexión a internet, etc.
  - Si tratamos de manejar todo esto con setState, la lógica se vuelve confusa y difícil de mantener.
- **Escalabilidad**
  - En apps grandes, el estado no afecta a un solo widget, sino a múltiples pantallas y componentes.
  - Pasar los datos manualmente entre widgets (lo que se conoce como prop drilling) se convierte en un caos.
- **Separación de responsabilidades**
  - Si mezclamos lógica de negocio con UI, cada widget termina siendo enorme y poco reutilizable.
  - Los gestores de estado nos permiten mantener la lógica en un lugar y la UI en otro, haciendo el código más limpio y fácil de extender.
- **Reactividad**
  - Queremos que la app responda automáticamente cuando cambie un dato (ejemplo: mostrar "Sesión expirada" si el token del usuario deja de ser válido).
  - Los gestores de estado implementan mecanismos reactivos que notifican a los widgets cuando deben actualizarse.
- **Testabilidad**
  - Es más fácil probar la lógica de negocio cuando está separada del UI.

- Con un gestor de estado podemos simular eventos y comprobar los resultados.s.

👉 **En resumen:**
Los gestores de estado en Flutter son fundamentales porque permiten construir apps mantenibles, escalables, reactivas y fáciles de probar. Sin ellos, el código se vuelve rápidamente inmanejable, incluso en aplicaciones de tamaño medio.

Flutter no impone una única forma de manejar el estado. Existen múltiples enfoques, que varían en simplicidad, escalabilidad y robustez. A continuación, revisamos los más usados:

4. 1. setState() → Estado Local

Es la forma más sencilla y directa. Se usa cuando el estado afecta solo a un widget o a un número reducido de widgets dentro de la misma pantalla.

Ejemplo: Contador básico con setState

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

class Contador extends StatefulWidget {
  const Contador({super.key});

  @override
  State<Contador> createState() => _ContadorState();
}

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("setState Demo")),
      body: Center(
        child: Text("Valor: $valor", style: const TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: incrementar,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

✅ **Ventajas**: fácil de implementar, ideal para prototipos o apps pequeñas.
⚠️ **Limitaciones**: se complica en proyectos medianos/grandes, ya que la lógica queda acoplada al

widget.

4. 2. Provider → Estado Global

La biblioteca provider es una biblioteca oficial de la comunidad Flutter que implementa el patrón ChangeNotifier. Permite exponer datos a toda la app sin necesidad de pasarlos manualmente entre widgets (lo que se conoce como prop drilling).

Ejemplo: Definir modelo de estado

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

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

  void incrementar() {
    _valor++;
    notifyListeners(); // Notifica a los widgets consumidores
  }
}
```

Integración en la app

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'contador_model.dart'; // tu clase de estado

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(home: HomePage());
  }
}

class HomePage extends StatelessWidget {
  const HomePage({super.key});
```

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

    return Scaffold(
      appBar: AppBar(title: const Text("Provider Demo")),
      body: Center(
        child: Text("Valor: ${contador.valor}", style: const
TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: contador.incrementar,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

✅ Ventajas: sencillo, escalable, oficial y muy usado.
⚠️ Limitaciones: el ChangeNotifier puede volverse difícil de mantener si el estado crece demasiado.

4. 3. Riverpod → Moderno, escalable y desacoplado

La biblioteca flutter_riverpod es la evolución de Provider, más seguro en tiempo de compilación, desacoplado y con menos boilerplate.

Ejemplo: Definir Provider con StateNotifier

```
import 'package:flutter_riverpod/flutter_riverpod.dart';

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

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

Uso en un ConsumerWidget
import 'package:flutter/material.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';
import 'contador_provider.dart';
```

```
class Home extends ConsumerWidget {
  const Home({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final valor = ref.watch(contadorProvider);

    return Scaffold(
      appBar: AppBar(title: const Text("Riverpod Demo")),
      body: Center(
        child: Text("Contador: $valor", style: const
TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () =>
ref.read(contadorProvider.notifier).incrementar(),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

✅ **Ventajas:** más moderno, desacoplado, testable, muy usado en proyectos grandes.
⚠️ **Limitaciones:** curva de aprendizaje un poco más alta que Provider.

5. DISEÑO RESPONSIVE Y ADAPTABILIDAD EN FLUTTER

En el desarrollo móvil y multiplataforma, uno de los mayores retos es que la aplicación se vea y funcione correctamente en diferentes dispositivos: móviles pequeños, tablets grandes e incluso pantallas web o de escritorio.

Flutter nos facilita este trabajo con una serie de herramientas que permiten crear interfaces adaptables y flexibles que se ajustan automáticamente al entorno.

📌 **Conceptos Clave de Responsividad**

🔹 **MediaQuery**

Permite acceder a información del dispositivo: tamaño de pantalla, orientación, densidad de píxeles, etc.

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

👉 Útil para ajustar márgenes, fuentes, tamaños o decidir si mostrar un layout móvil o tablet.

🔹 **LayoutBuilder**

Evalúa el espacio disponible en un widget y renderiza diferentes layouts según el ancho/alto.

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

👉 Ideal cuando queremos que un mismo widget cambie según el espacio que tenga (no solo según la pantalla completa).

🔹 **Expanded y Flexible**

Distribuyen espacio dentro de Row o Column.

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

👉 Permiten construir layouts que se adaptan sin necesidad de tamaños fijos.

📱 **Ejemplo Práctico: App Responsiva**

Objetivo: mostrar una lista vertical en móvil y un grid de dos columnas en tablet.

```
class ResponsiveLayout extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('App Responsiva')),
      body: LayoutBuilder(
        builder: (context, constraints) {
          if (constraints.maxWidth > 600) {
            // Vista para tablets
            return GridView.builder(
              gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 2,
              ),
              itemBuilder: (context, index) => ItemWidget(index),
```

```
            );
          } else {
            // Vista para móviles
            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

🔹 Breakpoints personalizados

Definimos reglas estándar para separar móvil, tablet y escritorio:

```
bool isMobile(BuildContext context) => MediaQuery.of(context).size.width < 600;
bool isTablet(BuildContext context) => MediaQuery.of(context).size.width >= 600 &&
MediaQuery.of(context).size.width < 1200;
bool isDesktop(BuildContext context) => MediaQuery.of(context).size.width >= 1200;
```

🔹 OrientationBuilder

Detecta cambios de orientación del dispositivo (vertical ↔ horizontal).

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

🔹 FractionallySizedBox

Permite asignar un porcentaje del espacio disponible.

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

✅ Buenas Prácticas en Diseño Responsivo

✔ Evitar dimensiones fijas (width: 300) → preferir MediaQuery, Expanded o Flexible.
✔ Testear en múltiples dispositivos → usar emuladores de distintos tamaños y, si es posible, probar en hardware real.
✔ Priorizar layouts adaptativos → que se reorganicen en lugar de simplemente escalar.
✔ Usar breakpoints claros (móvil < 600, tablet 600-1200, web > 1200).

📊 Ejemplo Final: App de Noticias Responsiva

- 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
  }
}
```

📊 Código Completo – Layout Responsivo Realista

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

void main() {
  runApp(MaterialApp(
    home: ResponsiveLayout(),
  ));
}

class ResponsiveLayout extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    final screenWidth = MediaQuery.of(context).size.width;

    if (screenWidth < 600) {
      // 📱 Móvil
      return Scaffold(
        appBar: AppBar(title: Text("App Responsiva - Móvil")),
        body: ListView.builder(
          itemCount: 20,
          itemBuilder: (context, index) => ListTile(
            leading: Icon(Icons.article),
            title: Text("Noticia $index"),
            subtitle: Text("Resumen breve de la noticia $index"),
          ),
        ),
      );
    } else if (screenWidth < 1200) {
      // 📲 Tablet
      return Scaffold(
        appBar: AppBar(title: Text("App Responsiva - Tablet")),
        body: GridView.builder(
          gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 2, // 2 columnas
            childAspectRatio: 3 / 2,
          ),
          itemCount: 20,
          itemBuilder: (context, index) => Card(
            margin: EdgeInsets.all(8),
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                Icon(Icons.article, size: 40, color: Colors.green),
                SizedBox(height: 10),
                Text("Noticia $index", style: TextStyle(fontWeight:
FontWeight.bold)),
                Text("Resumen breve..."),
              ],
            ),
          ),
        ),
      );
    } else {
      // 🖥️ Escritorio
      return Scaffold(
        appBar: AppBar(title: Text("App Responsiva - Escritorio")),
        body: Row(
          children: [
            // Sidebar
            Container(
              width: 250,
              color: Colors.grey[200],
              child: ListView(
                children: [
                  DrawerHeader(
```

```
                    child: Text("Menú", style: TextStyle(fontSize: 20, fontWeight:
FontWeight.bold)),
                  ),
                  ListTile(leading: Icon(Icons.home), title: Text("Inicio")),
                  ListTile(leading: Icon(Icons.category), title: Text("Categorías")),
                  ListTile(leading: Icon(Icons.settings), title:
Text("Configuración")),
                ],
              ),
            ),
            // Contenido principal
            Expanded(
              child: GridView.builder(
                gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                  crossAxisCount: 3, // 3 columnas
                  childAspectRatio: 4 / 3,
                ),
                itemCount: 30,
                itemBuilder: (context, index) => Card(
                  margin: EdgeInsets.all(10),
                  child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: [
                      Icon(Icons.article, size: 50, color: Colors.orange),
                      SizedBox(height: 10),
                      Text("Noticia $index", style: TextStyle(fontWeight:
FontWeight.bold)),
                      Padding(
                        padding: const EdgeInsets.all(8.0),
                        child: Text("Contenido adaptado para pantallas grandes"),
                      ),
                    ],
                  ),
                ),
              ),
            ),
          ],
        ),
      );
    }
  }
}
```

📝 Explicación

- Usamos MediaQuery para detectar el ancho.
- Definimos breakpoints:
  - <600px ⟶ ListView simple (móvil).
  - 600–1200px ⟶ Grid con 2 columnas (tablet).
  - >1200px ⟶ Sidebar + Grid con 3 columnas (escritorio).
- El layout cambia automáticamente al redimensionar la ventana (útil si pruebas en web de Flutter).

6. RECURSOS RECOMENDADOS PARA APRENDER FLUTTER

📚 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-ZoE6kzN1t9lfV9aYkRFYhwyj

📦 Paquetes y bibliotecas

- 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

- Ejemplos de Apps en GitHub
  🔗 https://github.co/flutter/samples