📚 Guía Completa de Flutter - PARTE 2

Continuación: Navegación, Formularios, Provider y Ejercicios

---

7. 🗺️ Navegación entre Pantallas

El Concepto: La Pila de Platos

Flutter maneja las pantallas como una **pila de platos** en el fregadero:

- **Push (empujar):** Pones un plato nuevo ENCIMA. Ahora ves el nuevo, pero el viejo sigue debajo.
- **Pop (quitar):** Quitas el plato de arriba. Vuelves a ver el que estaba debajo.

```
Inicio -> (push) -> Detalle -> (push) -> Configuración
                                  <- (pop) <-
      <- (pop) <-
```

Navigator.push: Ir a una Pantalla

```
// Desde un botón:
ElevatedButton(
  onPressed: () {
    Navigator.push(
      context, // El "contexto" actual (dónde estás)
      MaterialPageRoute(
        builder: (context) => SegundaPantalla(), // La pantalla a la que vas
      ),
    );
```

```
  },
  child: Text("Ir a Segunda Pantalla"),
)
```

¿Qué es `context`?
Es como la “dirección” de dónde estás en el árbol de widgets. Flutter lo necesita para saber desde dónde navegas.

¿Qué es `MaterialPageRoute`?
Es la “ruta” que Flutter usa para ir a la nueva pantalla con una animación bonita (deslizamiento).

Navigator.pop: Volver Atrás

```
ElevatedButton(
  onPressed: () {
    Navigator.pop(context); // Vuelve a la pantalla anterior
  },
  child: Text("Volver"),
)
```

⚠ **IMPORTANTE:** Si haces `pop` en la primera pantalla (cuando no hay nada debajo), ¡cierras la app!

Pasar Datos entre Pantallas

Método 1: Constructor (El más común)

Pantalla de Destino:

```
class DetallePage extends StatelessWidget {
  final String titulo; // Dato que recibimos
  final int precio;

  // Constructor: obligamos a recibir estos datos
```

```
  const DetallePage({
    super.key,
    required this.titulo,
    required this.precio,
  });

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(titulo)), // Usamos el dato
      body: Center(
        child: Text("Precio: $precio €", style: TextStyle(fontSize: 24)),
      ),
    );
  }
}
```

Al Navegar:

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => DetallePage(
      titulo: "Café Latte",
      precio: 3,
    ),
  ),
);
```

Método 2: Recibir Datos al Volver (pop con resultado)

Pantalla que recibe el dato:

```
// Esperamos un resultado
```

```
final resultado = await Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => SeleccionarColor()),
);

print("Color seleccionado: $resultado");
```

Pantalla que devuelve el dato:

```
ElevatedButton(
  onPressed: () {
    Navigator.pop(context, "Azul"); // Devolvemos "Azul"
  },
  child: Text("Seleccionar Azul"),
)
```

---

8. 📝 Formularios y Validación

¿Por qué usar Form y no solo TextField?

TextField solo: Tienes que validar cada campo manualmente.
Form: Puedes validar TODOS los campos de golpe con una sola línea.

Estructura de un Formulario

```
class MiFormulario extends StatefulWidget {
  @override
  _MiFormularioState createState() => _MiFormularioState();
}

class _MiFormularioState extends State<MiFormulario> {
  // 1. La "llave" para controlar el formulario
```

```dart
final _formKey = GlobalKey<FormState>();

// 2. Controladores para leer los valores
final _nombreCtrl = TextEditingController();
final _emailCtrl = TextEditingController();

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Formulario")),
    body: Padding(
      padding: EdgeInsets.all(16),
      child: Form( // 3. Envolvemos todo en un Form
        key: _formKey, // Le asignamos la llave
        child: Column(
          children: [
            // 4. Usamos TextFormField (no TextField)
            TextFormField(
              controller: _nombreCtrl,
              decoration: InputDecoration(
                labelText: "Nombre",
                border: OutlineInputBorder(),
              ),
              // 5. La función de validación
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'El nombre es obligatorio'; // Mensaje de error
                }
                if (value.length < 3) {
                  return 'Mínimo 3 caracteres';
                }
                return null; // null = todo correcto
              },
            ),
```

```dart
SizedBox(height: 16),

TextFormField(
  controller: _emailCtrl,
  decoration: InputDecoration(
    labelText: "Email",
    border: OutlineInputBorder(),
  ),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'El email es obligatorio';
    }
    if (!value.contains('@')) {
      return 'Debe contener @';
    }
    return null;
  },
),

SizedBox(height: 24),

// 6. Botón que valida
ElevatedButton(
  onPressed: () {
    // Validamos TODOS los campos de golpe
    if (_formKey.currentState!.validate()) {
      // Si llegamos aquí, TODO es válido
      print("Nombre: ${_nombreCtrl.text}");
      print("Email: ${_emailCtrl.text}");

      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Formulario válido ✓')),
      );
```

```
                }
              },
              child: Text('Enviar'),
            ),
          ],
        ),
      ),
    ),
  );
 }

 @override
 void dispose() {
  // 7. IMPORTANTE: Limpiar los controladores al salir
  _nombreCtrl.dispose();
  _emailCtrl.dispose();
  super.dispose();
 }
}
```

¿Qué hace cada parte?

1. **`GlobalKey<FormState>`:** Es como un “mando a distancia” para controlar el formu lario.
2. **`TextEditingController`:** Sirve para leer el texto que escribe el usuario.
3. **`Form`:** Agrupa todos los campos.
4. **`TextFormField`:** Como TextField, pero con validación integrada.
5. **`validator`:** Función que devuelve:
    - `null` si todo está bien
    - Un `String` con el mensaje de error si algo falla
6. **`_formKey.currentState!.validate()`:** Ejecuta TODOS los validators de golpe. Dev uelve `true` si todos son válidos.
7. **`dispose()`:** Limpia la memoria. Siempre hazlo con los controladores.

---

9. 🧠 Gestión de Estado con Provider

El Problema que Resuelve Provider

Imagina que tienes un carrito de compra. Necesitas mostrar el número de items en:

- La pantalla principal
- El menú lateral
- La barra superior

Sin Provider: Tendrías que pasar la variable del carrito de pantalla en pantalla (un lío).
Con Provider: El carrito está en un "lugar central" y TODAS las pantallas pueden acceder a él.

Analogía: La Casa Inteligente

- **El Modelo (ChangeNotifier):** Es el cuadro eléctrico de la casa. Aquí están los inter ruptores (datos).
- **notifyListeners():** Es como tocar un timbre 🔔 que avisa a toda la casa: "¡He cambi ado algo!"
- **context.watch:** Es como mirar la TV 📺. Si el canal cambia, lo ves al instante.
- **context.read:** Es como el mando a distancia 🎮. Lo usas para cambiar el canal, per o no necesitas estar mirando la TV.

Paso 1: Instalar Provider

En `pubspec.yaml`:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.0
```

Luego ejecuta: `flutter pub get`

Paso 2: Crear el Modelo

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

class CarritoModel extends ChangeNotifier {
  // Datos privados (nadie puede modificarlos directamente)
  final List<String> _productos = [];

  // Getter público (solo lectura)
  List<String> get productos => _productos;

  // Getter calculado
  int get totalItems => _productos.length;

  // Método para añadir
  void agregar(String producto) {
    _productos.add(producto);
    notifyListeners(); // 🔔 ¡IMPORTANTE! Avisa a las pantallas
  }

  // Método para quitar
  void quitar(String producto) {
    _productos.remove(producto);
    notifyListeners(); // 🔔 Avisa
  }

  // Método para vaciar
  void vaciar() {
    _productos.clear();
    notifyListeners(); // 🔔 Avisa
  }
}
```

¿Por qué `notifyListeners()`?

Sin esto, los datos cambiarían pero la pantalla NO se actualizaría. Es como cambiar el can al de la TV sin que nadie se entere.

Paso 3: Inyectar el Provider en main.dart

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

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => CarritoModel(), // Creamos el modelo
      child: MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: TiendaPage(),
    );
  }
}
```

¿Qué hace `ChangeNotifierProvider`?
Hace que el `CarritoModel` esté disponible para TODAS las pantallas de la app.

Paso 4: Consumir el Provider

Opción A: `context.watch` (Para MOSTRAR datos)

Úsalo cuando quieras que el widget se redibuje si el modelo cambia.

```dart
class ResumenCarrito extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Escuchamos el modelo
    final carrito = context.watch<CarritoModel>();

    return Container(
      padding: EdgeInsets.all(16),
      child: Text(
        "Items en carrito: ${carrito.totalItems}",
        style: TextStyle(fontSize: 20),
      ),
    );
  }
}
```

¿Qué pasa?
Cada vez que llames a `notifyListeners()` en el modelo, este widget se redibujará automáticamente.

Opción B: `context.read` (Para EJECUTAR acciones)

Úsalo dentro de botones o funciones. NO redibuja el widget.

```dart
class BotonAgregar extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        // Ejecutamos una acción (no necesitamos redibujar este botón)
        context.read<CarritoModel>().agregar("Manzana");
      },
      child: Text("Añadir Manzana"),
    );
```

```
  }
}
```

¿Cuándo usar watch vs read?

Situación	Usar
Mostrar un dato que puede cambiar	`context.watch`
Botón que ejecuta una acción	`context.read`
Dentro de `build()` para mostrar	`context.watch`
Dentro de `onPressed`, `onTap`, etc.	`context.read`

---

10. 🌐 Conexión a Internet (HTTP y JSON)

Conceptos Clave

HTTP: El "idioma" que usan las apps para hablar con servidores.
JSON: El formato de texto en que vienen los datos (como un diccionario/mapa).
Future: Una "promesa" de que un dato llegará en el futuro (porque internet tarda).
async/await: Palabras mágicas para esperar datos sin congelar la app.

Paso 1: Instalar el paquete http

En `pubspec.yaml`:

```
dependencies:
  http: ^1.1.0
```

Paso 2: Hacer una Petición GET

```
import 'dart:convert'; // Para jsonDecode
```

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

// Función asíncrona (devuelve un Future)
Future<List<dynamic>> obtenerPosts() async {
  // 1. Crear la URL
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');

  // 2. Hacer la petición (esperamos con await)
  final respuesta = await http.get(url);

  // 3. Verificar si fue exitosa (código 200 = OK)
  if (respuesta.statusCode == 200) {
    // 4. Convertir el texto JSON a una lista de mapas
    return jsonDecode(respuesta.body);
  } else {
    // 5. Si falló, lanzar un error
    throw Exception('Error al cargar datos');
  }
}
```

¿Qué es `async`?
Marca la función como “asíncrona” (que tarda tiempo).

¿Qué es `await`?
“Espera aquí hasta que llegue el resultado, pero sin congelar la app”.

¿Qué es `jsonDecode`?
Convierte el texto JSON en una estructura de Dart (List o Map).

Paso 3: Mostrar los Datos con FutureBuilder

`FutureBuilder` es un widget que:

1. Muestra un spinner mientras carga
2. Muestra un error si falla
3. Muestra los datos cuando llegan

```dart
class PaginaAPI extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Posts desde API")),
      body: FutureBuilder<List<dynamic>>(
        future: obtenerPosts(), // La función que trae los datos
        builder: (context, snapshot) {
          // snapshot contiene el estado actual

          // Caso 1: Todavía cargando
          if (snapshot.connectionState == ConnectionState.waiting) {
            return Center(child: CircularProgressIndicator());
          }

          // Caso 2: Hubo un error
          else if (snapshot.hasError) {
            return Center(
              child: Text("Error: ${snapshot.error}"),
            );
          }

          // Caso 3: Datos listos
          else if (snapshot.hasData) {
            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']),
                );
```

```
            },
          );
        }

        // Caso 4: No hay datos
        else {
          return Center(child: Text("No hay datos"));
        }
      },
    ),
  );
 }
}
```

Crear un Modelo de Datos (Buena Práctica)

En lugar de usar `Map<String, dynamic>`, crea una clase:

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

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

 // Factory constructor para crear desde JSON
 factory Post.fromJson(Map<String, dynamic> json) {
  return Post(
   id: json['id'],
   title: json['title'],
   body: json['body'],
  );
 }
}
```

```dart
// Ahora la función devuelve List<Post>
Future<List<Post>> obtenerPosts() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
  final respuesta = await http.get(url);

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

---

11. 📦 Listas Dinámicas

ListView.builder: Para Listas Largas

¿Por qué NO usar Column?
Si tienes 1000 elementos, Column los crea TODOS de golpe (lento y consume mucha me moria).

¿Por qué SÍ usar ListView.builder?
Solo crea los elementos que se ven en pantalla (eficiente).

```dart
final List<String> nombres = ['Ana', 'Beto', 'Carla', 'Dani', 'Elena'];

ListView.builder(
  itemCount: nombres.length, // Cuántos elementos hay
  itemBuilder: (context, index) {
    // Esta función se llama para cada elemento VISIBLE
    // index es la posición (0, 1, 2...)
```

```
    return ListTile(
      leading: CircleAvatar(
        child: Text(nombres[index][0]), // Primera letra
      ),
      title: Text(nombres[index]),
      subtitle: Text('Posición: $index'),
      trailing: Icon(Icons.arrow_forward),
      onTap: () {
        print('Tocaste a ${nombres[index]}');
      },
    );
  },
)
```

GridView.builder: Para Cuadrículas

```
GridView.builder(
  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
    crossAxisCount: 2, // 2 columnas
    crossAxisSpacing: 10, // Espacio horizontal
    mainAxisSpacing: 10, // Espacio vertical
  ),
  itemCount: 20,
  itemBuilder: (context, index) {
    return Container(
      color: Colors.blue[100 * (index % 9)],
      child: Center(child: Text('Item $index')),
    );
  },
)
```

Ahora continuaré con los ejercicios resueltos en detalle extremo en la siguiente parte…