📃 Guía Completa de Flutter - PARTE 4: EJERCICIOS FINALES

Ejercicios 4 y 5 Explicados en Detalle

✅ EJERCICIO 4: Formularios Accesibles

🎯 Objetivo del Ejercicio

Crear un formulario que valide un email y muestre errores de forma **accesible** (no solo con color rojo).

📚 Conceptos que Aprenderás

- Por qué la accesibilidad es importante
- Cómo mostrar errores sin depender del color
- Validación de formularios

🌐 ¿Por qué Accesibilidad?

El Problema:
Muchas apps solo ponen el borde rojo cuando un campo tiene error. Pero:

- Las personas daltónicas no ven el rojo
- Las personas con baja visión pueden no notar el cambio de color
- Los lectores de pantalla no “leen” colores

La Solución:
Usar **3 canales de comunicación**:

1. **Color** (para quien lo vea)
2. **Icono** (visual pero no depende del color)
3. **Texto** (explícito y leído por lectores de pantalla)

Código Completo Explicado

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

void main() {
  runApp(const MaterialApp(home: FormPage()));
}

// StatefulWidget porque el estado del error cambia
class FormPage extends StatefulWidget {
  const FormPage({super.key});

  @override
  State<FormPage> createState() => _FormPageState();
}

class _FormPageState extends State<FormPage> {
  // Controlador para leer el texto del campo
  final _emailCtrl = TextEditingController();

  // Variable de estado: mensaje de error
  // Si es null, no hay error
  // Si tiene texto, ese es el mensaje a mostrar
  String? mensajeError;

  // Función que valida el email
  void validar() {
    setState(() {
      // Leemos el texto del campo
      final texto = _emailCtrl.text;

      // Validación 1: ¿Está vacío?
      if (texto.isEmpty) {
        mensajeError = "El correo no puede estar vacío";
```

```
      return; // Salimos de la función
    }

    // Validación 2: ¿Tiene @?
    if (!texto.contains('@')) {
      mensajeError = "El correo necesita una '@'";
      return;
    }

    // Validación 3: ¿Tiene punto después de @?
    final partes = texto.split('@');
    if (partes.length != 2 || !partes[1].contains('.')) {
      mensajeError = "Formato de correo inválido";
      return;
    }

    // Si llegamos aquí, todo está bien
    mensajeError = null;

    // Mostramos un mensaje de éxito
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('✅ Email válido'),
        backgroundColor: Colors.green,
      ),
    );
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Formulario Accesible')),
    body: Padding(
```

```
padding: const EdgeInsets.all(16),
child: Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    // Etiqueta del campo
    const Text(
      'Correo electrónico',
      style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold),
    ),
    const SizedBox(height: 8),

    // Campo de texto
    TextField(
      controller: _emailCtrl,
      decoration: InputDecoration(
        border: const OutlineInputBorder(),
        hintText: 'ejemplo@correo.com',

        // Si hay error, ponemos el borde rojo
        // (pero NO es la única indicación)
        errorBorder: mensajeError != null
            ? const OutlineInputBorder(
                borderSide: BorderSide(color: Colors.red, width: 2),
              )
            : null,
      ),
      // Validar al cambiar el texto (opcional)
      onChanged: (_) {
        // Si había un error, lo limpiamos al empezar a escribir
        if (mensajeError != null) {
          setState(() => mensajeError = null);
        }
      },
    ),
```

```
const SizedBox(height: 10),

// ========== SECCIÓN DE ERROR ACCESIBLE ==========
// Solo se muestra si hay error
if (mensajeError != null)
  Container(
    padding: const EdgeInsets.all(12),
    decoration: BoxDecoration(
      color: Colors.red[50], // Fondo rojo suave
      borderRadius: BorderRadius.circular(8),
      border: Border.all(color: Colors.red, width: 1),
    ),
    child: Row(
      children: [
        // 1. ICONO (visual, no depende del color)
        const Icon(
          Icons.error_outline,
          color: Colors.red,
          size: 24,
        ),
        const SizedBox(width: 12),

        // 2. TEXTO EXPLÍCITO (leído por lectores de pantalla)
        Expanded(
          child: Text(
            mensajeError!,
            style: const TextStyle(
              color: Colors.red,
              fontWeight: FontWeight.bold,
            ),
          ),
        ),
      ],
```

```
            ),
          ),
          // ================================================

          const SizedBox(height: 20),

          // Botón de validar
          SizedBox(
            width: double.infinity,
            child: ElevatedButton(
              onPressed: validar,
              child: const Text('Validar'),
            ),
          ),
        ],
      ),
    ),
  );
 }

 @override
 void dispose() {
  // IMPORTANTE: Limpiar el controlador al salir
  _emailCtrl.dispose();
  super.dispose();
 }
}
```

🔍 Explicación Detallada

¿Qué es `String?` (con interrogación)?
Significa que la variable puede ser `String` o `null`. Si fuera solo `String`, tendría que tener un valor siempre.

¿Por qué usar `if (mensajeError != null)` en el widget?

Es una forma de mostrar/ocultar widgets condicionalmente. Si `mensajeError` es `null`, el Co ntainer de error no se construye.

¿Qué hace `Expanded` en el Row del error?
Hace que el texto ocupe todo el espacio disponible. Sin esto, si el mensaje es largo, se sal dría de la pantalla.

¿Por qué `dispose()`?
Los `TextEditingController` consumen memoria. Si no los limpias con `dispose()`, pueden ca usar “memory leaks” (fugas de memoria).

🎓 Resumen de Accesibilidad

Reglas de Oro:

1. ❌ NUNCA uses SOLO el color para indicar algo
2. ✅ SIEMPRE combina: Color + Icono + Texto
3. ✅ Usa textos claros y descriptivos
4. ✅ Asegúrate de que haya suficiente contraste

Ejemplo de MALA accesibilidad:

```
// Solo cambia el color del borde (mal)
TextField(
  decoration: InputDecoration(
    border: OutlineInputBorder(
      borderSide: BorderSide(color: hayError ? Colors.red : Colors.grey),
    ),
  ),
)
```

Ejemplo de BUENA accesibilidad:

```
// Color + Icono + Texto (bien)
if (hayError)
  Row(
```

```
    children: [
      Icon(Icons.error, color: Colors.red),
      Text("Error: El campo está vacío"),
    ],
  )
```

✅ EJERCICIO 5: Pokedex (API + GridView + Navegación)

🎯 Objetivo del Ejercicio

Crear una Pokedex que:

- Cargue datos de una API real (PokeAPI)
- Muestre los Pokemon en una cuadrícula
- Permita ver el detalle de cada Pokemon

📚 Conceptos que Aprenderás

- Consumir una API REST
- Parsear JSON
- Usar `GridView.builder`
- Combinar navegación con datos de API

PASO 1: El Modelo de Datos

```
// Clase que representa un Pokemon
class Pokemon {
  final String name;  // Nombre del Pokemon
  final String url;   // URL de la API con más info

  // Constructor
```

```dart
  Pokemon({required this.name, required this.url});

  // Getter calculado: extrae el ID de la URL
  // Ejemplo: "https://pokeapi.co/api/v2/pokemon/25/" → "25"
  String get id {
    // Dividimos la URL por "/"
    final partes = url.split('/');
    // El ID está en la posición 6
    return partes[6];
  }

  // Getter calculado: URL de la imagen
  String get imageUrl {
    return "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/$id.png";
  }

  // Factory constructor: crea un Pokemon desde JSON
  factory Pokemon.fromJson(Map<String, dynamic> json) {
    return Pokemon(
      name: json['name'],
      url: json['url'],
    );
  }
}
```

¿Qué es un `factory constructor`?
Es un constructor especial que puede devolver una instancia existente o crear una nueva. Aquí lo usamos para convertir JSON en un objeto Pokemon.

¿Por qué getters calculados?
En lugar de guardar el ID y la URL de la imagen en variables, los calculamos cuando se necesitan. Ahorra memoria.

PASO 2: La Pantalla Principal (Pokedex)

```dart
import 'dart:convert'; // Para jsonDecode
import 'package:http/http.dart' as http;
import 'package:flutter/material.dart';

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

  @override
  State<PokedexPage> createState() => _PokedexPageState();
}

class _PokedexPageState extends State<PokedexPage> {
  // Lista de Pokemon (vacía al principio)
  List<Pokemon> pokemons = [];

  // Estado de carga
  bool cargando = true;

  @override
  void initState() {
    super.initState();
    // initState se ejecuta UNA VEZ cuando se crea el widget
    // Aquí cargamos los datos
    cargarPokemons();
  }

  // Función asíncrona para cargar los Pokemon
  Future<void> cargarPokemons() async {
    try {
      // 1. Hacemos la petición HTTP
      final respuesta = await http.get(
        Uri.parse('https://pokeapi.co/api/v2/pokemon?limit=20'),
```

```
    );

    // 2. Verificamos que fue exitosa
    if (respuesta.statusCode == 200) {
      // 3. Parseamos el JSON
      final data = jsonDecode(respuesta.body);

      // 4. Extraemos la lista de resultados
      final lista = data['results'] as List;

      // 5. Convertimos cada elemento JSON en un Pokemon
      setState(() {
        pokemons = lista.map((json) => Pokemon.fromJson(json)).toList();
        cargando = false; // Ya no estamos cargando
      });
    } else {
      // Si el código no es 200, hubo un error
      throw Exception('Error al cargar Pokemon');
    }
  } catch (e) {
    // Capturamos cualquier error
    setState(() {
      cargando = false;
    });

    // Mostramos un mensaje de error
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: $e')),
    );
  }
}

@override
Widget build(BuildContext context) {
```

```
return Scaffold(
  appBar: AppBar(
    title: const Text("Pokedex"),
    backgroundColor: Colors.red,
  ),
  body: cargando
      // Mientras carga, mostramos un spinner
      ? const Center(child: CircularProgressIndicator())

      // Cuando termina de cargar, mostramos la cuadrícula
      : GridView.builder(
          // Configuración de la cuadrícula
          gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 2, // 2 columnas
            crossAxisSpacing: 10, // Espacio horizontal entre celdas
            mainAxisSpacing: 10, // Espacio vertical entre celdas
            childAspectRatio: 1, // Relación ancho/alto (1 = cuadrado)
          ),

          // Padding alrededor de toda la cuadrícula
          padding: const EdgeInsets.all(10),

          // Número de elementos
          itemCount: pokemons.length,

          // Constructor de cada celda
          itemBuilder: (context, index) {
            final poke = pokemons[index];

            return GestureDetector(
              // Al tocar, navegamos al detalle
              onTap: () {
                Navigator.push(
                  context,
```

```
        MaterialPageRoute(
          builder: (_) => DetallePokemon(pokemon: poke),
        ),
      );
    },

    child: Card(
      elevation: 4, // Sombra
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(12),
      ),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          // Imagen del Pokemon
          Image.network(
            poke.imageUrl,
            height: 100,
            width: 100,
            // Mientras carga la imagen, mostramos un spinner
            loadingBuilder: (context, child, loadingProgress) {
              if (loadingProgress == null) return child;
              return const CircularProgressIndicator();
            },
            // Si falla la carga, mostramos un icono
            errorBuilder: (context, error, stackTrace) {
              return const Icon(Icons.error, size: 50);
            },
          ),

          const SizedBox(height: 8),

          // Nombre del Pokemon
          Text(
```

```
                    poke.name.toUpperCase(),
                    style: const TextStyle(
                      fontWeight: FontWeight.bold,
                      fontSize: 14,
                    ),
                    textAlign: TextAlign.center,
                  ),

                  // ID del Pokemon
                  Text(
                    '#${poke.id}',
                    style: TextStyle(
                      color: Colors.grey[600],
                      fontSize: 12,
                    ),
                  ),
                ],
              ),
            ),
          );
        },
      ),
    );
  }
}
```

🔍 Explicación Detallada

¿Qué es `initState()`?
Es un método que se ejecuta UNA SOLA VEZ cuando el widget se crea. Es el lugar perfecto para cargar datos iniciales.

¿Por qué `async` y `await`?

- `async`: Marca la función como asíncrona (que tarda tiempo)
- `await`: "Espera aquí hasta que llegue el resultado, pero sin congelar la app"

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

Ejemplo:

```
String json = '{"name": "pikachu", "id": 25}';
Map<String, dynamic> mapa = jsonDecode(json);
print(mapa['name']); // "pikachu"
```

¿Qué hace `.map()`?
Transforma cada elemento de una lista.

Ejemplo:

```
List<int> numeros = [1, 2, 3];
List<int> dobles = numeros.map((n) => n * 2).toList();
// dobles = [2, 4, 6]
```

En nuestro caso:

```
lista.map((json) => Pokemon.fromJson(json)).toList()
```

Convierte cada elemento JSON en un objeto Pokemon.

¿Qué es `GestureDetector`?
Un widget invisible que detecta gestos (toques, deslizamientos, etc.). Aquí lo usamos para hacer que la Card sea tocable.

PASO 3: La Pantalla de Detalle

```
class DetallePokemon extends StatelessWidget {
  final Pokemon pokemon; // Pokemon que recibimos

  const DetallePokemon({super.key, required this.pokemon});
```

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(pokemon.name.toUpperCase()),
      backgroundColor: Colors.red,
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          // Imagen grande
          Image.network(
            pokemon.imageUrl,
            height: 200,
            width: 200,
          ),

          const SizedBox(height: 20),

          // Nombre
          Text(
            pokemon.name.toUpperCase(),
            style: const TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.bold,
            ),
          ),

          const SizedBox(height: 10),

          // ID
          Text(
```

```
            'Pokemon #${pokemon.id}',
            style: TextStyle(
              fontSize: 20,
              color: Colors.grey[600],
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

🎓 Resumen del Ejercicio 5

Flujo completo:

1. La app inicia → `initState()` llama a `cargarPokemons()`
2. `cargarPokemons()` hace una petición HTTP a la PokeAPI
3. Recibe un JSON con una lista de Pokemon
4. Convierte cada Pokemon JSON en un objeto `Pokemon`
5. Actualiza el estado con `setState()`
6. Flutter redibuja la pantalla mostrando el `GridView`
7. Usuario toca un Pokemon → Navega a `DetallePokemon`

Conceptos clave:

- **HTTP GET:** Pedir datos a un servidor
- **JSON:** Formato de texto para intercambiar datos
- **async/await:** Esperar datos sin congelar la app
- **GridView.builder:** Cuadrícula eficiente para muchos elementos
- **Navegación con datos:** Pasar objetos entre pantallas

🎯 RESUMEN FINAL DE TODOS LOS EJERCICIOS

Ejercicio 1: Provider

- **Aprendiste:** Gestión de estado global
- **Cuándo usarlo:** Cuando varios widgets necesitan acceder a los mismos datos
- **Clave:** `notifyListeners()`, `context.watch`, `context.read`

Ejercicio 2: Navegación

- **Aprendiste:** Ir entre pantallas y volver
- **Cuándo usarlo:** Apps con múltiples pantallas
- **Clave:** `Navigator.push`, `Navigator.pop`

Ejercicio 3: Animaciones

- **Aprendiste:** Animaciones implícitas
- **Cuándo usarlo:** Transiciones suaves entre estados
- **Clave:** Widgets `Animated...`, operador ternario

Ejercicio 4: Formularios

- **Aprendiste:** Validación y accesibilidad
- **Cuándo usarlo:** Formularios de login, registro, etc.
- **Clave:** `Form`, `TextFormField`, `validator`, accesibilidad

Ejercicio 5: API

- **Aprendiste:** Consumir APIs REST
- **Cuándo usarlo:** Apps que necesitan datos de internet
- **Clave:** `http.get`, `jsonDecode`, `FutureBuilder`, `GridView.builder`

📝 CONSEJOS PARA EL EXAMEN

1. **Lee bien el enunciado:** Identifica qué te piden (navegación, formularios, API, etc.)
2. **Empieza por la estructura:** `Scaffold` → `AppBar` → `body`
3. **Usa `const` siempre que puedas:** Mejora el rendimiento
4. **No olvides `setState()`:** Si cambias una variable y no ves el cambio, falta `setState()`
5. **Provider:** `watch` para mostrar, `read` para acciones
6. **Errores comunes:**
    - Olvidar `notifyListeners()` en Provider
    - Usar `watch` en un botón (ineficiente)
    - No hacer `dispose()` de los controladores
    - Olvidar `await` en funciones asíncronas

¡Buena suerte en el examen! 🚀