Manual de Desarrollo de Aplicaciones con Flutter

Capítulo 1: Introducción a Flutter y Configuración del Entorno

1.1 ¿Qué es Flutter y por qué usarlo?

Flutter es un SDK (Software Development Kit) de código abierto desarrollado por Google para la creación de aplicaciones compiladas de forma nativa para dispositivos móviles (Android e iOS), web y escritorio (Windows, macOS, Linux) a partir de una única base de código. Su principal atractivo radica en su capacidad para ofrecer un rendimiento cercano al nativo y una experiencia de usuario altamente personalizable, gracias a su propio motor de renderizado Skia.

La elección de Flutter para el desarrollo de aplicaciones multiplataforma se justifica por varias razones clave:

- **Desarrollo rápido**: La función de Hot Reload y Hot Restart permite a los desarrolladores ver los cambios en tiempo real, lo que acelera significativamente el ciclo de desarrollo y depuración. Esto se traduce en una mayor productividad y la capacidad de iterar rápidamente sobre el diseño y la funcionalidad de la aplicación.
- **Rendimiento nativo**: A diferencia de otros frameworks multiplataforma que utilizan puentes JavaScript para comunicarse con los componentes nativos, Flutter compila directamente a código ARM nativo para móviles, y a WebAssembly para la web, lo que resulta en un rendimiento excepcional y una experiencia de usuario fluida y sin interrupciones. El motor de renderizado Skia, utilizado por Google Chrome y Android, permite a Flutter dibujar la interfaz de usuario directamente en la pantalla, sin depender de los widgets nativos del sistema operativo.
- **Interfaz de usuario hermosa y expresiva**: Flutter ofrece un conjunto rico y personalizable de widgets preconstruidos que siguen las directrices de Material Design (para Android) y Cupertino (para iOS). Además, permite la creación de widgets personalizados con gran facilidad, lo que brinda a los diseñadores y desarrolladores una libertad creativa sin precedentes para construir interfaces de usuario visualmente atractivas y únicas. La arquitectura basada en widgets de

Flutter facilita la composición de interfaces complejas a partir de componentes más pequeños y reutilizables.

- **Productividad mejorada**: Con una única base de código, los equipos de desarrollo pueden dirigirse a múltiples plataformas, lo que reduce el tiempo y el esfuerzo necesarios para mantener y actualizar las aplicaciones. Esto no solo optimiza los recursos, sino que también asegura una experiencia de usuario consistente en todos los dispositivos y plataformas. La documentación exhaustiva y una comunidad activa también contribuyen a una curva de aprendizaje más suave y a la resolución eficiente de problemas.

- **Crecimiento y soporte de Google**: Flutter cuenta con el respaldo de Google, lo que garantiza su evolución continua y la integración de nuevas características y mejoras. La comunidad de desarrolladores de Flutter es vibrante y en constante crecimiento, lo que se traduce en una gran cantidad de recursos, paquetes y soporte disponibles.

En resumen, Flutter se ha consolidado como una opción robusta y eficiente para el desarrollo de aplicaciones multiplataforma, ofreciendo un equilibrio óptimo entre rendimiento, flexibilidad y productividad. Su enfoque en la experiencia del desarrollador y la calidad de la interfaz de usuario lo convierten en una herramienta poderosa para construir aplicaciones modernas y atractivas.

1.2 Configuración del entorno de desarrollo (SDK, IDE, emuladores)

Para comenzar a desarrollar con Flutter, es esencial configurar correctamente el entorno de desarrollo. Este proceso implica la instalación del SDK de Flutter, un entorno de desarrollo integrado (IDE) y la configuración de emuladores o dispositivos físicos para probar las aplicaciones.

1.2.1 Instalación del SDK de Flutter

El SDK de Flutter contiene todas las herramientas necesarias para compilar aplicaciones Flutter. Los pasos generales para su instalación son los siguientes:

1. **Descargar el SDK**: Visita la página oficial de Flutter (flutter.dev/docs/get-started/install) y descarga la versión estable del SDK para tu sistema operativo (Windows, macOS, Linux).
2. **Extraer el archivo**: Descomprime el archivo ZIP o tar.gz descargado en una ubicación de tu elección. Se recomienda una ruta sencilla como `C:\flutter` en Windows o `/Users/tu_usuario/flutter` en macOS/Linux.
3. **Añadir Flutter al PATH**: Para poder ejecutar los comandos de Flutter desde cualquier terminal, es necesario añadir la ruta del directorio `bin` del SDK a la

variable de entorno PATH de tu sistema. Los pasos varían ligeramente según el sistema operativo:
    - **Windows**: Busca "Editar las variables de entorno del sistema", haz clic en "Variables de entorno", selecciona "Path" en "Variables del sistema" y haz clic en "Editar". Añade la ruta completa a la carpeta `flutter\bin`.
    - **macOS/Linux**: Abre tu archivo de configuración de shell (por ejemplo, `.bashrc`, `.zshrc` o `.profile`) y añade la siguiente línea, reemplazando `[RUTA_A_FLUTTER]` con la ruta donde descomprimiste el SDK: `bash export PATH="$PATH:[RUTA_A_FLUTTER]/flutter/bin"` Luego, guarda el archivo y ejecuta `source ~/.bashrc` (o el archivo correspondiente) para aplicar los cambios.
4. **Verificar la instalación**: Abre una nueva terminal y ejecuta el comando `flutter doctor`. Este comando verifica tu entorno y te informa si falta alguna dependencia o si hay algún problema de configuración. Deberías ver un informe detallado con el estado de tu instalación de Flutter, Android Studio, Xcode (para iOS en macOS), y las herramientas de desarrollo web.

1.2.2 Elección y configuración del IDE

Aunque puedes desarrollar con Flutter en cualquier editor de texto, se recomienda encarecidamente el uso de un IDE con soporte completo para Flutter y Dart. Los más populares son Android Studio y Visual Studio Code.

- **Android Studio**: Es el IDE oficial para el desarrollo de Android y ofrece una integración robusta con Flutter. Para configurarlo:

    1. Descarga e instala Android Studio desde developer.android.com/studio.
    2. Abre Android Studio y ve a `File > Settings > Plugins` (o `Android Studio > Preferences > Plugins` en macOS).
    3. Busca e instala los plugins de "Flutter" y "Dart". Esto instalará automáticamente las dependencias necesarias.
    4. Reinicia Android Studio.

- **Visual Studio Code (VS Code)**: Es un editor de código ligero pero potente, muy popular entre los desarrolladores de Flutter. Para configurarlo:

    1. Descarga e instala VS Code desde code.visualstudio.com.
    2. Abre VS Code y ve a la vista de Extensiones (Ctrl+Shift+X o Cmd+Shift+X).
    3. Busca e instala las extensiones de "Flutter" y "Dart".
    4. Reinicia VS Code.

Ambos IDEs ofrecen características como autocompletado de código, depuración, refactorización y la capacidad de ejecutar aplicaciones en emuladores o dispositivos físicos directamente desde el IDE.

1.2.3 Configuración de emuladores y dispositivos físicos

Para probar tus aplicaciones Flutter, necesitarás un dispositivo Android o iOS, o un emulador/simulador.

- **Emulador de Android (Android Studio)**:
    1. En Android Studio, ve a `Tools > Device Manager` (o `AVD Manager` en versiones anteriores).
    2. Haz clic en "Create Virtual Device" y selecciona un dispositivo (por ejemplo, Pixel 5).
    3. Elige una imagen de sistema (versión de Android) y descárgala si es necesario.
    4. Configura las opciones del emulador y haz clic en "Finish".
    5. Puedes iniciar el emulador desde el Device Manager.
- **Simulador de iOS (macOS con Xcode)**:
    1. Si estás en macOS, descarga e instala Xcode desde la Mac App Store. Esto incluye las herramientas de línea de comandos y el simulador de iOS.
    2. Abre Xcode, ve a `Xcode > Preferences > Locations` y asegúrate de que las "Command Line Tools" estén seleccionadas.
    3. Puedes iniciar el simulador de iOS desde `Xcode > Open Developer Tool > Simulator`.
- **Dispositivos físicos**: Para probar en un dispositivo físico, habilita la "Depuración USB" en las opciones de desarrollador de tu dispositivo Android o iOS. Conecta el dispositivo a tu computadora y asegúrate de que sea reconocido por `flutter devices` en la terminal.

Una vez que el entorno esté configurado, podrás crear y ejecutar tu primera aplicación Flutter. Asegúrate de ejecutar `flutter doctor` regularmente para verificar el estado de tu configuración y resolver cualquier problema que pueda surgir.

1.3 Estructura básica de un proyecto Flutter

Cuando creas un nuevo proyecto Flutter, se genera una estructura de directorios estándar que organiza los archivos de tu aplicación. Comprender esta estructura es fundamental para navegar por tu proyecto y saber dónde colocar cada tipo de archivo.

La estructura típica de un proyecto Flutter es la siguiente:

```
mi_aplicacion_flutter/
├── .dart_tool/
├── .idea/
├── android/
├── ios/
├── lib/
│   ├── main.dart
│   └── ... (otros archivos Dart de tu aplicación)
├── linux/
├── macos/
├── test/
├── web/
├── windows/
├── .gitignore
├── .metadata
├── pubspec.yaml
├── README.md
└── ...
```

A continuación, se describe el propósito de los directorios y archivos más importantes:

- **`.dart_tool/`**: Contiene archivos generados automáticamente por las herramientas de Dart y Flutter. Generalmente, no necesitas interactuar directamente con este directorio.
- **`.idea/`**: Contiene archivos de configuración específicos para el IDE IntelliJ IDEA/ Android Studio. Si usas VS Code, este directorio podría no existir o ser reemplazado por `.vscode/`.
- **`android/`**: Este directorio contiene el proyecto Android nativo de tu aplicación. Si necesitas añadir funcionalidades específicas de Android, configurar permisos o integrar librerías nativas de Android, lo harás en este directorio. Flutter utiliza este proyecto para compilar tu aplicación para la plataforma Android.
- **`ios/`**: Similar al directorio `android/`, este contiene el proyecto iOS nativo de tu aplicación. Aquí es donde configurarás permisos de iOS, añadirás librerías de CocoaPods o realizarás configuraciones específicas para la plataforma iOS. Flutter lo utiliza para compilar tu aplicación para iOS.
- **`lib/`**: Este es el corazón de tu aplicación Flutter. Contiene todo el código fuente de tu aplicación escrito en Dart. Dentro de `lib/`, encontrarás:
    - **`main.dart`**: Es el punto de entrada de tu aplicación. Aquí es donde se inicia la ejecución de tu aplicación Flutter. Típicamente, este archivo contiene la función `main()` y la clase principal de tu aplicación (por ejemplo, `MyApp`).

    - **Otros archivos `.dart`**: A medida que tu aplicación crece, organizarás tu código en múltiples archivos Dart dentro de este directorio, separando la lógica, la interfaz de usuario, los modelos, los servicios, etc., en módulos más pequeños y manejables.

- **`linux/`, `macos/`, `web/`, `windows/`**: Estos directorios contienen los proyectos específicos para las plataformas de escritorio (Linux, macOS, Windows) y web, respectivamente. Al igual que con `android/` e `ios/`, si necesitas realizar configuraciones o añadir código específico para estas plataformas, lo harás en sus respectivos directorios.

- **`test/`**: Contiene los archivos de prueba de tu aplicación. Flutter promueve el desarrollo basado en pruebas, y este directorio es donde escribirás tus pruebas unitarias, de widgets y de integración para asegurar la calidad y el correcto funcionamiento de tu código.

- **`.gitignore`**: Un archivo estándar de Git que especifica qué archivos y directorios deben ser ignorados por el sistema de control de versiones (Git). Contiene entradas para archivos generados, dependencias y otros elementos que no deben ser versionados.

- **`pubspec.yaml`**: Este es un archivo muy importante en un proyecto Flutter. Es un archivo de configuración YAML que define los metadatos de tu proyecto, como el nombre, la descripción y la versión. Más importante aún, es donde declaras las dependencias de tu proyecto (paquetes de Dart y Flutter que tu aplicación utiliza), los activos (imágenes, fuentes, etc.) y otras configuraciones específicas de Flutter. Cada vez que añades una nueva dependencia, debes añadirla aquí y luego ejecutar `flutter pub get`.

- **`README.md`**: Un archivo Markdown que proporciona una descripción general de tu proyecto, instrucciones de configuración, cómo ejecutar la aplicación y cualquier otra información relevante para otros desarrolladores o para ti mismo en el futuro.

Comprender esta estructura te permitirá organizar tu código de manera eficiente y colaborar con otros desarrolladores de forma más efectiva. A medida que avancemos en el manual, exploraremos en detalle cómo interactuar con cada una de estas partes del proyecto.

1.4 Hot Reload y Hot Restart

Una de las características más aclamadas y productivas de Flutter es su capacidad de **Hot Reload** (recarga en caliente) y **Hot Restart** (reinicio en caliente). Estas funcionalidades revolucionan el ciclo de desarrollo al permitir a los desarrolladores ver

los cambios en el código casi instantáneamente, sin perder el estado de la aplicación en el caso del Hot Reload.

1.4.1 Hot Reload

El Hot Reload es una característica que inyecta los cambios de código fuente actualizados en la aplicación en ejecución sin necesidad de reiniciar la aplicación por completo. Esto significa que puedes modificar tu código Dart, guardar los cambios y ver cómo se reflejan en la interfaz de usuario en cuestión de segundos, manteniendo el estado actual de la aplicación. Por ejemplo, si estás en una pantalla específica de tu aplicación y realizas un cambio en el diseño de un widget, el Hot Reload aplicará ese cambio sin navegar de nuevo a la pantalla de inicio o perder los datos que hayas introducido.

¿Cómo funciona el Hot Reload?

Cuando realizas un cambio en tu código y lo guardas, el motor de Flutter:

1. **Detecta los cambios**: El IDE (Android Studio o VS Code) o la línea de comandos detecta que se han guardado cambios en los archivos Dart.
2. **Inyecta el código**: El motor de Flutter inyecta el nuevo código en la Dart Virtual Machine (VM) que está ejecutando la aplicación. Esto es posible porque Dart es un lenguaje que soporta "hot reloading" a nivel de lenguaje.
3. **Reconstruye el árbol de widgets**: Flutter reconstruye el árbol de widgets de la aplicación con el código actualizado. Los widgets que han sido modificados o que dependen de los cambios se redibujan, mientras que el estado de los widgets que no han sido afectados se mantiene.

Ventajas del Hot Reload:

- **Velocidad**: Permite iterar rápidamente sobre el diseño y la funcionalidad, lo que acelera significativamente el proceso de desarrollo.
- **Mantiene el estado**: No pierdes el estado actual de la aplicación, lo que es invaluable para depurar interfaces de usuario complejas o flujos de trabajo de varias pantallas.
- **Productividad**: Reduce el tiempo de espera y la frustración, permitiendo a los desarrolladores mantenerse en el "flujo" de trabajo.

Limitaciones del Hot Reload:

Aunque el Hot Reload es extremadamente útil, hay ciertas situaciones en las que no puede aplicarse y se requiere un Hot Restart:

- **Cambios en `pubspec.yaml`**: Si añades o eliminas dependencias en tu archivo `pubspec.yaml`, el Hot Reload no funcionará. Necesitarás ejecutar `flutter pub get` y luego un Hot Restart.
- **Cambios en el código nativo**: Si modificas el código Java/Kotlin (Android) o Swift/Objective-C (iOS) en los directorios `android/` o `ios/`, el Hot Reload no podrá inyectar esos cambios. Esto requiere un Hot Restart o una reconstrucción completa.
- **Cambios en el `main()` o `initState()`**: Algunos cambios estructurales importantes, como la modificación de la función `main()` o la lógica dentro de los métodos `initState()` de los `StatefulWidget`s, pueden requerir un Hot Restart para que los cambios surtan efecto correctamente.

1.4.2 Hot Restart

El Hot Restart, a diferencia del Hot Reload, reinicia completamente la aplicación. Esto significa que la aplicación vuelve a su estado inicial, como si la hubieras lanzado por primera vez. Todos los estados de los widgets se pierden y la aplicación se reconstruye desde cero.

¿Cuándo usar Hot Restart?

- Cuando el Hot Reload no es suficiente para aplicar los cambios (por ejemplo, después de modificar `pubspec.yaml` o código nativo).
- Cuando necesitas reiniciar la aplicación a su estado inicial para probar un flujo de usuario desde el principio.
- Cuando experimentas comportamientos inesperados después de múltiples Hot Reloads y quieres asegurarte de que la aplicación está en un estado limpio.

¿Cómo funciona el Hot Restart?

El Hot Restart detiene la Dart VM actual y la reinicia, cargando la versión más reciente del código de la aplicación. Aunque es más lento que el Hot Reload, sigue siendo significativamente más rápido que una reconstrucción completa de la aplicación, ya que no implica recompilar el código nativo.

En la práctica, los desarrolladores de Flutter utilizan el Hot Reload la mayor parte del tiempo para iterar rápidamente en la UI y la lógica de la aplicación, reservando el Hot Restart para aquellos casos en los que los cambios son más profundos o cuando necesitan restablecer el estado de la aplicación.

Capítulo 2: Widgets y Diseño de UI Avanzado

Flutter se basa en el concepto de widgets para construir interfaces de usuario. Todo en Flutter es un widget: desde un simple texto o un botón, hasta el layout de toda la pantalla. Comprender cómo funcionan los widgets y cómo combinarlos para crear diseños complejos y responsivos es fundamental para desarrollar aplicaciones Flutter robustas y visualmente atractivas.

2.1 Fundamentos de los Widgets (StatelessWidget, StatefulWidget)

En Flutter, los widgets son los bloques de construcción fundamentales de la interfaz de usuario. Describen cómo debe verse y comportarse una parte de la UI en un momento dado. Existen dos tipos principales de widgets en Flutter, que se distinguen por su capacidad para manejar el estado:

2.1.1 StatelessWidget

Un `StatelessWidget` es un widget que no tiene estado mutable. Esto significa que su configuración y apariencia son inmutables una vez que se construyen. Son ideales para partes de la UI que no cambian con el tiempo o en respuesta a la interacción del usuario. Piensa en ellos como componentes estáticos que solo dependen de los datos que se les proporcionan.

Características clave de un `StatelessWidget`:

- **Inmutabilidad**: Una vez que se crea un `StatelessWidget`, sus propiedades no pueden cambiar. Si necesitas que la UI cambie, deberás crear un nuevo `StatelessWidget` con las nuevas propiedades.
- **No tiene estado interno**: No almacena ningún estado que pueda cambiar durante la vida útil del widget.
- **Método `build()`**: La lógica principal de un `StatelessWidget` reside en su método `build()`. Este método toma un `BuildContext` como argumento y devuelve un árbol de widgets que describe la parte de la UI que este widget representa.

Ejemplo de `StatelessWidget`:

Imagina un widget simple para mostrar un título y un subtítulo:

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

class TituloConSubtitulo extends StatelessWidget {
  final String titulo;
```

```
  final String subtitulo;

  const TituloConSubtitulo({
    Key? key,
    required this.titulo,
    required this.subtitulo,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(
          titulo,
          style: const TextStyle(
            fontSize: 24,
            fontWeight: FontWeight.bold,
          ),
        ),
        Text(
          subtitulo,
          style: const TextStyle(
            fontSize: 16,
            color: Colors.grey,
          ),
        ),
      ],
    );
  }
}

// Uso en otra parte de la aplicación:
// TituloConSubtitulo(titulo: 'Bienvenido', subtitulo: 'Explora
nuestra app')
```

En este ejemplo, `TituloConSubtitulo` es un `StatelessWidget` porque su título y subtítulo no cambiarán una vez que se haya creado. Si quisiéramos cambiar el texto, tendríamos que crear una nueva instancia de `TituloConSubtitulo`.

2.1.2 StatefulWidget

Un `StatefulWidget` es un widget que tiene estado mutable. Esto significa que su configuración puede cambiar durante la vida útil del widget, y la UI puede redibujarse en respuesta a esos cambios. Son adecuados para partes de la UI que necesitan actualizarse dinámicamente, como un contador, un checkbox, o un campo de entrada de texto.

Características clave de un `StatefulWidget`:

- **Estado mutable**: Almacena un estado que puede cambiar con el tiempo.
- **Clase `State` separada**: Un `StatefulWidget` se compone de dos clases: el `StatefulWidget` en sí y una clase `State` asociada. La clase `StatefulWidget` es inmutable y define la configuración inicial, mientras que la clase `State` es mutable y contiene el estado del widget y la lógica para construir la UI.
- **Método `createState()`**: La clase `StatefulWidget` debe sobrescribir el método `createState()`, que devuelve una instancia de su clase `State` asociada.
- **Método `setState()`**: Para notificar a Flutter que el estado interno del widget ha cambiado y que la UI necesita ser redibujada, se utiliza el método `setState()`. Cuando se llama a `setState()`, Flutter marca el widget como "sucio" y lo reconstruye en el siguiente fotograma.

Ejemplo de `StatefulWidget`:

Considera un widget de contador simple que incrementa un número cada vez que se presiona un botón:

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

class ContadorWidget extends StatefulWidget {
  const ContadorWidget({Key? key}) : super(key: key);

  @override
  State<ContadorWidget> createState() => _ContadorWidgetState();
}

class _ContadorWidgetState extends State<ContadorWidget> {
  int _contador = 0;

  void _incrementarContador() {
    setState(() {
      _contador++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(
          'Contador: $_contador',
          style: const TextStyle(fontSize: 30),
        ),
```

```
        ElevatedButton(
          onPressed: _incrementarContador,
          child: const Text('Incrementar'),
        ),
      ],
    );
  }
}

// Uso en otra parte de la aplicación:
// ContadorWidget()
```

En este ejemplo, `ContadorWidget` es un `StatefulWidget` porque el valor de `_contador` cambia con la interacción del usuario. La clase `_ContadorWidgetState` mantiene el estado de `_contador` y el método `_incrementarContador` utiliza `setState()` para actualizar el valor y notificar a Flutter que debe redibujar el widget.

2.1.3 Cuándo usar cada uno

La elección entre `StatelessWidget` y `StatefulWidget` es fundamental y depende de si el widget necesita manejar un estado interno que pueda cambiar con el tiempo. Una buena regla general es:

- Utiliza `StatelessWidget` cuando la parte de la UI que estás construyendo no necesita cambiar después de su creación inicial. Son más ligeros y eficientes.
- Utiliza `StatefulWidget` cuando la parte de la UI necesita cambiar dinámicamente en respuesta a eventos (interacción del usuario, datos de red, temporizadores, etc.).

Es importante destacar que un `StatelessWidget` puede contener `StatefulWidget`s como parte de su árbol de widgets, y viceversa. La clave es identificar qué parte de la UI necesita mantener un estado mutable y encapsular esa lógica dentro de un `StatefulWidget`.

2.2 Layouts responsivos (MediaQuery, LayoutBuilder, AspectRatio)

El diseño responsivo es crucial en el desarrollo de aplicaciones modernas, ya que las aplicaciones se ejecutan en una amplia variedad de dispositivos con diferentes tamaños de pantalla, orientaciones y densidades de píxeles. Flutter proporciona varias herramientas y widgets para crear layouts que se adapten elegantemente a estas variaciones, asegurando una experiencia de usuario óptima en cualquier dispositivo.

2.2.1 MediaQuery

`MediaQuery` es un widget que proporciona información sobre el tamaño y la orientación de la pantalla del dispositivo, la densidad de píxeles, el relleno del sistema (como la barra de estado o la muesca) y otras características del entorno. Puedes acceder a esta información a través de `MediaQuery.of(context)`.

Propiedades útiles de `MediaQueryData`:

- `size`: El tamaño lógico de la pantalla (ancho y alto) en píxeles independientes del dispositivo (dp).
- `orientation`: La orientación de la pantalla (`Portrait` o `Landscape`).
- `padding`: El relleno del sistema en cada lado de la pantalla (por ejemplo, la barra de estado en la parte superior o la barra de navegación en la parte inferior).
- `devicePixelRatio`: La relación de píxeles del dispositivo.

Ejemplo de uso de `MediaQuery`:

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

class PantallaResponsiva extends StatelessWidget {
  const PantallaResponsiva({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final mediaQueryData = MediaQuery.of(context);
    final screenWidth = mediaQueryData.size.width;
    final screenHeight = mediaQueryData.size.height;
    final orientation = mediaQueryData.orientation;

    return Scaffold(
      appBar: AppBar(
        title: const Text('Diseño Responsivo con MediaQuery'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              'Ancho de pantalla: $
{screenWidth.toStringAsFixed(2)} dp',
              style: const TextStyle(fontSize: 20),
            ),
            Text(
              'Alto de pantalla: $
{screenHeight.toStringAsFixed(2)} dp',
              style: const TextStyle(fontSize: 20),
            ),
```

```
            Text(
              'Orientación: ${orientation ==
Orientation.portrait ? 'Vertical' : 'Horizontal'}',
              style: const TextStyle(fontSize: 20),
            ),
            SizedBox(height: 20),
            Container(
              width: screenWidth * 0.8, // 80% del ancho de la
pantalla
              height: orientation == Orientation.portrait ?
screenHeight * 0.3 : screenHeight * 0.6,
              color: Colors.blue,
              child: Center(
                child: Text(
                  'Contenedor Adaptativo',
                  style: TextStyle(color: Colors.white,
fontSize: 22),
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

En este ejemplo, el tamaño y la altura del `Container` se ajustan dinámicamente en función del ancho y la orientación de la pantalla, demostrando cómo `MediaQuery` permite adaptar la UI a diferentes tamaños de dispositivo.

2.2.2 LayoutBuilder

Mientras que `MediaQuery` proporciona información sobre el tamaño de la pantalla completa, `LayoutBuilder` es útil cuando necesitas adaptar un widget a las restricciones de tamaño de su padre. Esto es particularmente útil en escenarios donde un widget no ocupa toda la pantalla, sino solo una parte de ella, y necesitas que se adapte al espacio disponible.

`LayoutBuilder` es un widget que construye un árbol de widgets basado en las restricciones de tamaño de su padre. Proporciona un `BoxConstraints` que puedes usar para determinar el ancho y alto disponibles para el widget.

Ejemplo de uso de `LayoutBuilder`:

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

```dart
class AdaptarContenedor extends StatelessWidget {
  const AdaptarContenedor({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Diseño Responsivo con
LayoutBuilder'),
      ),
      body: Center(
        child: Container(
          width: 300, // Ancho fijo para el contenedor padre
          height: 300, // Alto fijo para el contenedor padre
          color: Colors.grey[300],
          child: LayoutBuilder(
            builder: (BuildContext context, BoxConstraints
constraints) {
              // Las restricciones son las del contenedor padre
(300x300)
              if (constraints.maxWidth > 200) {
                return Center(
                  child: Text(
                    'Ancho > 200: $
{constraints.maxWidth.toStringAsFixed(2)}',
                    style: const TextStyle(fontSize: 18, color:
Colors.black),
                  ),
                );
              } else {
                return Center(
                  child: Text(
                    'Ancho <= 200: $
{constraints.maxWidth.toStringAsFixed(2)}',
                    style: const TextStyle(fontSize: 18, color:
Colors.black),
                  ),
                );
              }
            },
          ),
        ),
      ),
    );
  }
}
```

En este ejemplo, el `LayoutBuilder` se utiliza dentro de un `Container` con un tamaño fijo. El texto dentro del `LayoutBuilder` cambia en función del `maxWidth` disponible para el `LayoutBuilder`, que en este caso es el ancho del `Container`

padre. Esto demuestra cómo `LayoutBuilder` permite crear widgets que se adaptan a las restricciones de espacio de sus padres, no solo a las de la pantalla completa.

2.2.3 AspectRatio

`AspectRatio` es un widget que intenta dimensionar a su hijo a una relación de aspecto específica. Es útil cuando quieres que un widget mantenga una proporción particular entre su ancho y su alto, independientemente del espacio disponible.

Propiedades de `AspectRatio`:

- `aspectRatio`: La relación de aspecto deseada (ancho / alto). Por ejemplo, `16 / 9` para una relación de aspecto de pantalla ancha, o `1 / 1` para un cuadrado.

Ejemplo de uso de `AspectRatio`:

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

class ContenedorConAspectRatio extends StatelessWidget {
  const ContenedorConAspectRatio({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Diseño Responsivo con AspectRatio'),
      ),
      body: Center(
        child: Container(
          width: double.infinity, // Ocupa todo el ancho
disponible
          height: 200, // Altura fija para el contenedor padre
          color: Colors.grey[300],
          child: AspectRatio(
            aspectRatio: 16 / 9, // Mantener una relación de
aspecto de 16:9
            child: Container(
              color: Colors.teal,
              child: const Center(
                child: Text(
                  'Widget con AspectRatio 16:9',
                  style: TextStyle(color: Colors.white,
fontSize: 20),
                ),
              ),
            ),
          ),
        ),
      ),
```

```
    );
  }
}
```

En este ejemplo, el `Container` interno se ajustará para mantener una relación de aspecto de 16:9, incluso si el `Container` padre tiene un ancho `double.infinity` y una altura fija. Flutter calculará el ancho o el alto del hijo para satisfacer la relación de aspecto dada, dentro de las restricciones del padre.

2.2.4 Otros widgets y técnicas para layouts responsivos

Además de `MediaQuery`, `LayoutBuilder` y `AspectRatio`, Flutter ofrece otros widgets y técnicas para construir interfaces responsivas:

- **`Expanded` y `Flexible`**: Utilizados dentro de `Row` y `Column` para distribuir el espacio disponible entre los hijos de manera flexible. `Expanded` fuerza a su hijo a llenar el espacio disponible, mientras que `Flexible` permite que su hijo se ajuste a su tamaño intrínseco pero con la capacidad de expandirse o contraerse.
- **`FittedBox`**: Escala y posiciona a su hijo dentro de sí mismo de acuerdo con un `BoxFit`.
- **`FractionallySizedBox`**: Dimensiona a su hijo a una fracción del tamaño disponible.
- **`OrientationBuilder`**: Similar a `LayoutBuilder`, pero reconstruye su hijo cuando la orientación del dispositivo cambia.
- **`CustomScrollView` y `Slivers`**: Para layouts de desplazamiento complejos que se adaptan a diferentes tamaños de pantalla y proporcionan efectos de desplazamiento personalizados.

Combinando estas herramientas, los desarrolladores pueden crear interfaces de usuario altamente adaptables que se ven y funcionan bien en una amplia gama de dispositivos, desde teléfonos pequeños hasta tabletas grandes y pantallas de escritorio.

2.3 Widgets de Material Design y Cupertino

Flutter proporciona dos conjuntos de widgets preconstruidos que implementan las directrices de diseño de las plataformas móviles más populares: Material Design para Android y Cupertino para iOS. Estos widgets permiten a los desarrolladores crear aplicaciones que se sienten nativas en cada plataforma, siguiendo sus respectivas convenciones de diseño y experiencia de usuario.

2.3.1 Material Design Widgets

Material Design es un sistema de diseño creado por Google que proporciona un conjunto de directrices para la interfaz de usuario, la interacción y el movimiento en Android, así como en otras plataformas. Flutter ofrece una rica biblioteca de widgets que implementan estos principios, lo que facilita la creación de aplicaciones con una apariencia moderna y consistente.

Para utilizar los widgets de Material Design, tu aplicación debe estar envuelta en un `MaterialApp`.

Ejemplos de Material Design Widgets comunes:

- **`Scaffold`**: Proporciona una estructura básica para la pantalla de una aplicación Material Design, incluyendo `AppBar`, `Drawer`, `SnackBar`, `FloatingActionButton`, `BottomNavigationBar`, etc. `dart Scaffold( appBar: AppBar(title: Text("Mi App Material")), body: Center(child: Text("Contenido")), floatingActionButton: FloatingActionButton(onPressed: () {}, child: Icon(Icons.add), ), );`
- **`AppBar`**: Una barra de aplicaciones Material Design que se muestra en la parte superior de la pantalla. Puede contener un título, acciones y un botón de navegación.
- **`Text`**: Muestra una cadena de texto con opciones de estilo.
- **`ElevatedButton`**, **`TextButton`**, **`OutlinedButton`**: Diferentes tipos de botones con estilos Material Design.
- **`Card`**: Un panel Material Design con esquinas redondeadas y una elevación sutil, utilizado para agrupar contenido relacionado.
- **`ListView`**, **`GridView`**: Widgets de desplazamiento para mostrar listas y cuadrículas de elementos.
- **`TextField`**: Un campo de entrada de texto Material Design.
- **`SnackBar`**: Un mensaje ligero que se muestra brevemente en la parte inferior de la pantalla.
- **`Drawer`**: Un panel de navegación que se desliza desde el borde de la pantalla.

- **`BottomNavigationBar`**: Una barra de navegación que se muestra en la parte inferior de la pantalla, ideal para la navegación de nivel superior.

La mayoría de los widgets de Material Design se encuentran en el paquete `package:flutter/material.dart`.

2.3.2 Cupertino Widgets

Para las aplicaciones que se ejecutan en iOS, Flutter ofrece un conjunto de widgets que imitan la apariencia y el comportamiento de las directrices de interfaz humana de Apple, conocidas como Cupertino Design. Esto permite a los desarrolladores crear aplicaciones que se sienten nativas para los usuarios de iOS.

Para utilizar los widgets de Cupertino, tu aplicación debe estar envuelta en un `CupertinoApp`.

Ejemplos de Cupertino Widgets comunes:

- **`CupertinoPageScaffold`**: Proporciona una estructura básica para una pantalla de aplicación iOS, similar a `Scaffold`. `dart CupertinoPageScaffold( navigationBar: CupertinoNavigationBar(middle: Text("Mi App Cupertino")), child: Center(child: Text("Contenido")), );`
- **`CupertinoNavigationBar`**: Una barra de navegación estilo iOS que se muestra en la parte superior de la pantalla.
- **`CupertinoButton`**: Un botón estilo iOS.
- **`CupertinoTextField`**: Un campo de entrada de texto estilo iOS.
- **`CupertinoSwitch`**: Un interruptor de encendido/apagado estilo iOS.
- **`CupertinoAlertDialog`**: Una alerta estilo iOS.
- **`CupertinoTabBar`**: Una barra de pestañas estilo iOS en la parte inferior de la pantalla.

Los widgets de Cupertino se encuentran en el paquete `package:flutter/cupertino.dart`.

2.3.3 Adaptación de la UI a la plataforma (Platform-aware UI)

Una de las ventajas de Flutter es la capacidad de crear una UI que se adapte automáticamente a la plataforma en la que se ejecuta. Esto se puede lograr de varias maneras:

- **Uso condicional de widgets**: Puedes usar `Theme.of(context).platform` para determinar la plataforma actual y renderizar el widget apropiado. Por ejemplo: `dart if (Theme.of(context).platform == TargetPlatform.iOS) { return CupertinoButton(...); } else { return ElevatedButton(...); }`

- **Paquetes de terceros**: Existen paquetes como `flutter_platform_widgets` que simplifican la creación de widgets que se adaptan automáticamente a la plataforma, proporcionando un widget unificado que renderiza el equivalente de Material o Cupertino según sea necesario.

- **Diseño unificado con Material Design**: Muchos desarrolladores optan por usar Material Design en ambas plataformas (Android e iOS) para mantener una apariencia y experiencia de usuario consistentes en todas partes. Material Design es lo suficientemente flexible como para verse bien en iOS, y muchos usuarios están familiarizados con él.

La elección de utilizar Material Design, Cupertino, o una combinación de ambos, dependerá de los requisitos específicos del proyecto y de la experiencia de usuario deseada. Flutter ofrece la flexibilidad para implementar cualquiera de estas estrategias, permitiendo a los desarrolladores crear aplicaciones que se sientan naturales y atractivas en cualquier plataforma.

2.4 Widgets personalizados y animaciones implícitas

La flexibilidad de Flutter para crear interfaces de usuario no se limita a los widgets predefinidos. Los desarrolladores tienen la capacidad de construir sus propios widgets personalizados, lo que permite una libertad creativa ilimitada. Además, Flutter facilita la adición de animaciones sutiles y fluidas a la UI a través de animaciones implícitas, mejorando la experiencia del usuario sin la necesidad de un control de animación complejo.

2.4.1 Creación de Widgets Personalizados

Un widget personalizado es simplemente un `StatelessWidget` o un `StatefulWidget` que encapsula una combinación de otros widgets para crear un

componente de UI reutilizable y específico para tu aplicación. Esto promueve la modularidad, la legibilidad del código y la reutilización.

¿Por qué crear widgets personalizados?

- **Reutilización**: Evita la duplicación de código al encapsular lógica y UI comunes en un solo widget.
- **Modularidad**: Divide la UI en componentes más pequeños y manejables, lo que facilita el desarrollo y el mantenimiento.
- **Legibilidad**: Mejora la comprensión del código al dar nombres significativos a partes complejas de la UI.
- **Personalización**: Permite crear componentes de UI que se ajustan perfectamente a la identidad visual y funcional de tu aplicación.

Ejemplo de Widget Personalizado:

Supongamos que queremos crear una tarjeta de perfil de usuario que muestre una imagen, un nombre y un rol. Podríamos combinar varios widgets básicos para lograr esto:

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

class TarjetaPerfilUsuario extends StatelessWidget {
  final String nombre;
  final String rol;
  final String urlImagen;

  const TarjetaPerfilUsuario({
    Key? key,
    required this.nombre,
    required this.rol,
    required this.urlImagen,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      margin: const EdgeInsets.all(16.0),
      elevation: 8.0,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(15.0),
      ),
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            CircleAvatar(
```

```
              radius: 50,
              backgroundImage: NetworkImage(urlImagen),
            ),
            const SizedBox(height: 10),
            Text(
              nombre,
              style: const TextStyle(
                fontSize: 22,
                fontWeight: FontWeight.bold,
              ),
            ),
            const SizedBox(height: 5),
            Text(
              rol,
              style: TextStyle(
                fontSize: 16,
                color: Colors.grey[600],
              ),
            ),
            const SizedBox(height: 15),
            Row(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                IconButton(
                  icon: const Icon(Icons.email),
                  onPressed: () { /* Lógica para enviar email
*/ },
                ),
                IconButton(
                  icon: const Icon(Icons.phone),
                  onPressed: () { /* Lógica para llamar */ },
                ),
              ],
            ),
          ],
        ),
      ),
    );
  }
}

// Uso en otra parte de la aplicación:
// TarjetaPerfilUsuario(
//   nombre: 'Juan Pérez',
//   rol: 'Desarrollador Flutter',
//   urlImagen: 'https://example.com/avatar.jpg',
// )
```

Este `TarjetaPerfilUsuario` es un `StatelessWidget` que combina `Card`, `Padding`, `Column`, `CircleAvatar`, `Text`, `SizedBox` y `IconButton` para formar

un componente cohesivo y reutilizable. Si la información del perfil pudiera cambiar dinámicamente, haríamos que `TarjetaPerfilUsuario` fuera un `StatefulWidget` y manejaríamos el estado internamente o a través de un patrón de gestión de estado.

2.4.2 Animaciones Implícitas

Las animaciones son clave para una experiencia de usuario fluida y atractiva. Flutter ofrece un sistema de animación potente, y las animaciones implícitas son la forma más sencilla de añadir movimiento a tu UI. Un widget de animación implícita es un `StatefulWidget` que anima automáticamente los cambios en sus propiedades.

Flutter proporciona una serie de widgets de animación implícita "listos para usar" que facilitan la animación de propiedades comunes. Estos widgets tienen un prefijo `Animated`.

Ejemplos de Widgets de Animación Implícita:

- **`AnimatedContainer`**: Anima automáticamente los cambios en sus propiedades como `width`, `height`, `color`, `decoration`, `padding`, `margin`, `alignment`, etc.
- **`AnimatedOpacity`**: Anima los cambios en la opacidad de su hijo.
- **`AnimatedPositioned`**: Anima la posición de un widget dentro de un `Stack`.
- **`AnimatedCrossFade`**: Anima la transición entre dos widgets hijos, con un efecto de fundido cruzado.
- **`AnimatedSwitcher`**: Anima la transición entre un widget hijo y otro.

Todos los widgets `Animated` requieren una `duration` (duración) para la animación y, opcionalmente, una `curve` (curva de animación).

Ejemplo de `AnimatedContainer`:

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

class ContenedorAnimadoEjemplo extends StatefulWidget {
  const ContenedorAnimadoEjemplo({Key? key}) : super(key: key);

  @override
  State<ContenedorAnimadoEjemplo> createState() =>
_ContenedorAnimadoEjemploState();
}

class _ContenedorAnimadoEjemploState extends
State<ContenedorAnimadoEjemplo> {
  double _width = 50.0;
```

```
  double _height = 50.0;
  Color _color = Colors.green;
  BorderRadiusGeometry _borderRadius =
BorderRadius.circular(8.0);

  void _actualizarContenedor() {
    setState(() {
      final random = Random();

      _width = random.nextInt(300).toDouble() + 50;
      _height = random.nextInt(300).toDouble() + 50;
      _color = Color.fromRGBO(
        random.nextInt(256),
        random.nextInt(256),
        random.nextInt(256),
        1,
      );
      _borderRadius =
BorderRadius.circular(random.nextInt(100).toDouble());
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('AnimatedContainer Ejemplo'),
      ),
      body: Center(
        child: AnimatedContainer(
          width: _width,
          height: _height,
          decoration: BoxDecoration(
            color: _color,
            borderRadius: _borderRadius,
          ),
          duration: const Duration(seconds: 1),
          curve: Curves.fastOutSlowIn,
          child: const Center(
            child: Text(
              '¡Anímame!',
              style: TextStyle(color: Colors.white, fontSize:
20),
            ),
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _actualizarContenedor,
        child: const Icon(Icons.play_arrow),
      ),
    );
```

```
  }
}
```

En este ejemplo, cada vez que se presiona el `FloatingActionButton`, las propiedades `_width`, `_height`, `_color` y `_borderRadius` del `AnimatedContainer` cambian. Gracias a `AnimatedContainer`, Flutter anima automáticamente la transición entre los valores antiguos y nuevos de estas propiedades durante la `duration` especificada, creando un efecto visual suave y agradable.

Las animaciones implícitas son una excelente manera de añadir interactividad y pulido a tu aplicación sin la complejidad de los `AnimationController`s y `Tween`s, que se utilizan para animaciones más personalizadas y controladas.

2.5 Temas y estilos

La consistencia visual es un pilar fundamental en el diseño de aplicaciones, y Flutter ofrece un sistema robusto para gestionar temas y estilos en toda tu aplicación. Esto te permite definir una apariencia unificada para elementos como colores, tipografías, formas y elevaciones, y aplicarla de manera centralizada, facilitando cambios globales y manteniendo la coherencia de la marca.

2.5.1 `ThemeData` y `Theme` Widget

El corazón del sistema de temas de Flutter es la clase `ThemeData`. Esta clase encapsula todas las propiedades visuales de tu aplicación, como la paleta de colores (`primaryColor`, `accentColor`, `colorScheme`), la tipografía (`textTheme`), la forma de los botones (`buttonTheme`), la apariencia de los `AppBar`s (`appBarTheme`), y mucho más. Al definir un `ThemeData`, estás creando un "tema" para tu aplicación.

Para aplicar este tema a tu aplicación, utilizas el widget `Theme`. Típicamente, el `ThemeData` se define en el `MaterialApp` (o `CupertinoApp`) de tu aplicación, lo que hace que el tema esté disponible para todos los widgets descendientes en el árbol de widgets.

Ejemplo de definición de `ThemeData`:

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

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

class MyApp extends StatelessWidget {
```

```dart
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Mi Aplicación con Tema Personalizado',
      theme: ThemeData(
        primarySwatch:
Colors.blueGrey, // Color principal y sus matices
        colorScheme: ColorScheme.fromSwatch(primarySwatch:
Colors.blueGrey).copyWith(
          secondary: Colors.teal, // Color de acento
        ),
        fontFamily: 'Roboto', // Fuente predeterminada
        appBarTheme: const AppBarTheme(
          backgroundColor: Colors.blueGrey,
          foregroundColor: Colors.white, // Color del texto y
los iconos en el AppBar
          elevation: 4.0,
        ),
        textTheme: const TextTheme(
          headlineLarge: TextStyle(fontSize: 32.0, fontWeight:
FontWeight.bold, color: Colors.blueGrey),
          bodyLarge: TextStyle(fontSize: 18.0, color:
Colors.black87),
          bodyMedium: TextStyle(fontSize: 16.0, color:
Colors.black54),
        ),
        buttonTheme: ButtonThemeData(
          buttonColor: Colors.teal,
          textTheme: ButtonTextTheme.primary,
          shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(10.0)),
        ),
        cardTheme: CardTheme(
          elevation: 6.0,
          shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(12.0)),
          margin: const EdgeInsets.all(8.0),
        ),
      ),
      home: const HomePage(),
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

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

```
      appBar: AppBar(
        title: const Text('Página Principal'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              'Título Grande',
              style: Theme.of(context).textTheme.headlineLarge,
            ),
            const SizedBox(height: 20),
            Text(
              'Este es un texto de cuerpo.',
              style: Theme.of(context).textTheme.bodyLarge,
            ),
            const SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {},
              child: const Text('Botón Temático'),
            ),
            const SizedBox(height: 20),
            Card(
              child: Padding(
                padding: const EdgeInsets.all(16.0),
                child: Text(
                  'Tarjeta con estilo temático.',
                  style: Theme.of(context).textTheme.bodyMedium,
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

En este ejemplo, hemos definido un `ThemeData` personalizado para nuestra aplicación. Observa cómo los widgets como `AppBar`, `ElevatedButton` y `Card` adoptan automáticamente los estilos definidos en el tema sin necesidad de especificar sus propiedades individualmente. Para acceder a las propiedades del tema desde cualquier widget, utilizamos `Theme.of(context)`.

2.5.2 Temas Oscuros y Claros

Flutter facilita la implementación de temas oscuros (dark mode) y claros (light mode), permitiendo a los usuarios elegir su preferencia o que la aplicación se adapte a la

configuración del sistema operativo. Esto se logra configurando las propiedades `theme` y `darkTheme` en `MaterialApp`.

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

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Temas Oscuro y Claro',
      theme: ThemeData(
        brightness: Brightness.light, // Tema claro
        primarySwatch: Colors.blue,
        // ... otras configuraciones para el tema claro
      ),
      darkTheme: ThemeData(
        brightness: Brightness.dark, // Tema oscuro
        primarySwatch: Colors.indigo,
        // ... otras configuraciones para el tema oscuro
      ),
      themeMode: ThemeMode.system, // O ThemeMode.light,
ThemeMode.dark
      home: const HomePage(),
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Página Principal'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              '¡Hola, Mundo!',
              style: Theme.of(context).textTheme.headlineLarge,
            ),
            const SizedBox(height: 20),
```

```
            ElevatedButton(
              onPressed: () {},
              child: const Text('Botón'),
            ),
          ],
        ),
      ),
    );
  }
}
```

La propiedad `themeMode` en `MaterialApp` te permite controlar cuándo se aplica el tema oscuro:

- `ThemeMode.system`: La aplicación sigue la configuración de tema del sistema operativo (predeterminado).
- `ThemeMode.light`: Siempre usa el tema claro.
- `ThemeMode.dark`: Siempre usa el tema oscuro.

2.5.3 Estilos de Texto con `TextTheme`

El `textTheme` dentro de `ThemeData` es una propiedad crucial para definir un sistema de tipografía consistente en tu aplicación. Permite especificar estilos para diferentes tipos de texto (títulos, subtítulos, cuerpo de texto, etc.) de acuerdo con las directrices de Material Design.

```
textTheme: const TextTheme(
  displayLarge: TextStyle(fontSize: 57.0, fontWeight:
FontWeight.w400, letterSpacing: -0.25, height: 1.12),
  displayMedium: TextStyle(fontSize: 45.0, fontWeight:
FontWeight.w400, height: 1.16),
  displaySmall: TextStyle(fontSize: 36.0, fontWeight:
FontWeight.w400, height: 1.22),
  headlineLarge: TextStyle(fontSize: 32.0, fontWeight:
FontWeight.w400, height: 1.25),
  headlineMedium: TextStyle(fontSize: 28.0, fontWeight:
FontWeight.w400, height: 1.29),
  headlineSmall: TextStyle(fontSize: 24.0, fontWeight:
FontWeight.w400, height: 1.33),
  titleLarge: TextStyle(fontSize: 22.0, fontWeight:
FontWeight.w400, height: 1.27),
  titleMedium: TextStyle(fontSize: 16.0, fontWeight:
FontWeight.w500, letterSpacing: 0.15, height: 1.50),
  titleSmall: TextStyle(fontSize: 14.0, fontWeight:
FontWeight.w500, letterSpacing: 0.10, height: 1.43),
  bodyLarge: TextStyle(fontSize: 16.0, fontWeight:
FontWeight.w400, letterSpacing: 0.50, height: 1.50),
  bodyMedium: TextStyle(fontSize: 14.0, fontWeight:
```

```
FontWeight.w400, letterSpacing: 0.25, height: 1.43),
  bodySmall: TextStyle(fontSize: 12.0, fontWeight:
FontWeight.w400, letterSpacing: 0.40, height: 1.67),
  labelLarge: TextStyle(fontSize: 14.0, fontWeight:
FontWeight.w500, letterSpacing: 0.10, height: 1.43),
  labelMedium: TextStyle(fontSize: 12.0, fontWeight:
FontWeight.w500, letterSpacing: 0.50, height: 1.33),
  labelSmall: TextStyle(fontSize: 11.0, fontWeight:
FontWeight.w500, letterSpacing: 0.50, height: 1.45),
),
```

Al definir estos estilos una vez, puedes asegurarte de que todos los textos de tu aplicación sigan las mismas directrices de diseño, simplemente accediendo a ellos a través de `Theme.of(context).textTheme.nombreDelEstilo`.

2.5.4 Estilos Personalizados y Extensiones de Tema

Para escenarios más avanzados, donde necesitas propiedades de tema que no están cubiertas por `ThemeData` (por ejemplo, colores de marca específicos, gradientes, o estilos de sombra personalizados), puedes crear tus propias extensiones de tema. Esto se logra utilizando `ThemeExtension`.

1. **Define tu clase de extensión de tema:** ```dart import 'package:flutter/material.dart';

   @immutable class CustomAppColors extends ThemeExtension { const CustomAppColors({ required this.brandColor, required this.dangerColor, });

   final Color? brandColor; final Color? dangerColor;

   @override CustomAppColors copyWith({Color? brandColor, Color? dangerColor}) { return CustomAppColors(brandColor: brandColor ?? this.brandColor, dangerColor: dangerColor ?? this.dangerColor,); }

   @override CustomAppColors lerp(ThemeExtension? other, double t) { if (other is! CustomAppColors) { return this; } return CustomAppColors(brandColor: Color.lerp(brandColor, other.brandColor, t), dangerColor: Color.lerp(dangerColor, other.dangerColor, t),); }

   // Opcional: para depuración @override String toString() => 'CustomAppColors(brandColor: $brandColor, dangerColor: $dangerColor)'; } ```

2. **Añade tu extensión a `ThemeData`:** `dart MaterialApp( theme: ThemeData( // ... otras configuraciones de tema extensions: const <ThemeExtension<dynamic>>[ CustomAppColors( brandColor:`

`Color(0xFF1A73E8), // Un azul de marca dangerColor: Color(0xFFD93025), // Un rojo de peligro ), ], ), // ... );`

3. **Accede a tus propiedades personalizadas:** `dart // En cualquier widget que tenga acceso al BuildContext final customColors = Theme.of(context).extension<CustomAppColors>()!; Container(color: customColors.brandColor);`

Esta técnica permite una personalización profunda y organizada de los temas, asegurando que todos los aspectos visuales de tu aplicación puedan ser gestionados de manera centralizada y consistente. Al dominar el sistema de temas y estilos de Flutter, puedes crear aplicaciones con una identidad visual fuerte y fácil de mantener.

Capítulo 3: Gestión de Estado

La gestión de estado es uno de los conceptos más cruciales y a menudo desafiantes en el desarrollo de aplicaciones Flutter. El "estado" de una aplicación se refiere a cualquier dato que pueda cambiar con el tiempo y que afecte la interfaz de usuario. Una gestión de estado eficiente es vital para construir aplicaciones escalables, mantenibles y con un rendimiento óptimo. Flutter, al ser declarativo, requiere que el estado de la UI se reconstruya cada vez que los datos subyacentes cambian. Este capítulo explorará los conceptos básicos de la gestión de estado y varias soluciones populares en el ecosistema de Flutter.

3.1 Conceptos básicos de gestión de estado

Antes de sumergirnos en las diferentes soluciones, es fundamental comprender los conceptos subyacentes de la gestión de estado en Flutter.

3.1.1 ¿Qué es el estado?

En el contexto de Flutter, el estado se refiere a la información que puede cambiar durante la vida útil de un widget y que afecta la forma en que se dibuja la interfaz de usuario. Este estado puede ser de dos tipos principales:

- **Estado efímero (Ephemeral State)**: También conocido como estado local o UI state. Es el estado que se contiene completamente dentro de un solo widget y no necesita ser compartido con otros widgets. Por ejemplo, la posición actual de un `PageView`, si un `Checkbox` está marcado o no, o el texto actual en un `TextField`. Este tipo de estado se gestiona típicamente con un `StatefulWidget` y su método `setState()`.

- **Estado de la aplicación (App State)**: También conocido como estado compartido o global state. Es el estado que se comparte entre múltiples widgets, persiste entre sesiones de usuario, o se obtiene de fuentes externas como una base de datos o una API. Ejemplos incluyen los datos de un usuario autenticado, el contenido de un carrito de compras, o los resultados de una llamada a una API. Este tipo de estado requiere soluciones de gestión de estado más robustas para ser compartido y actualizado de manera eficiente a través de la aplicación.

3.1.2 La naturaleza declarativa de Flutter

Flutter es un framework declarativo. Esto significa que construyes tu UI describiendo cómo debe verse en un momento dado, en función de su estado actual. Cuando el estado cambia, Flutter reconstruye la parte afectada del árbol de widgets para reflejar el nuevo estado. A diferencia de los frameworks imperativos, donde manipulas directamente los elementos de la UI, en Flutter, simplemente declaras cómo debe ser la UI para un estado dado, y el framework se encarga de las actualizaciones.

3.1.3 El papel de `BuildContext`

El `BuildContext` es un identificador para la ubicación de un widget en el árbol de widgets. Cada widget tiene su propio `BuildContext`. Es crucial para la gestión de estado porque permite a los widgets acceder a los datos y servicios proporcionados por widgets ancestros en el árbol. Por ejemplo, `Theme.of(context)` utiliza el `BuildContext` para encontrar el `ThemeData` más cercano en el árbol.

3.1.4 El flujo de datos en Flutter

En Flutter, el flujo de datos es típicamente unidireccional, de arriba hacia abajo. Los datos fluyen desde los widgets padres a los hijos a través de los constructores. Cuando un widget hijo necesita comunicar un evento o un cambio de estado a su padre, lo hace a través de callbacks. Este patrón ayuda a mantener la previsibilidad y la depuración del flujo de datos.

3.1.5 La importancia de la inmutabilidad

Aunque el estado puede ser mutable, es una buena práctica en Flutter (y en Dart en general) favorecer la inmutabilidad siempre que sea posible. Cuando los objetos de estado son inmutables, es más fácil razonar sobre los cambios y evitar efectos secundarios inesperados. En lugar de modificar un objeto de estado existente, se crea una nueva instancia del objeto con los datos actualizados. Esto es especialmente relevante en soluciones de gestión de estado que se basan en la reconstrucción de widgets cuando las referencias de los objetos cambian.

Comprender estos conceptos básicos es el primer paso para elegir la solución de gestión de estado adecuada para tu aplicación. Las siguientes secciones explorarán las herramientas y patrones más comunes para manejar el estado en Flutter, desde las opciones más simples hasta las más avanzadas.

3.2 `setState` y `InheritedWidget`

Estas son las formas más fundamentales y nativas de Flutter para manejar el estado. Aunque `setState` es adecuado para el estado efímero, `InheritedWidget` es la base sobre la que se construyen muchas soluciones de gestión de estado más avanzadas para el estado de la aplicación.

3.2.1 `setState` para el estado efímero

Como se mencionó en el Capítulo 2, `setState` es el método principal para actualizar el estado de un `StatefulWidget`. Cuando se llama a `setState()`, Flutter marca el widget como "sucio" y programa una reconstrucción del widget y sus descendientes en el siguiente fotograma. Esto es ideal para el estado que solo afecta a un único widget o a un subárbol pequeño de widgets.

Cuándo usar `setState`:

- Cuando el estado es local y solo afecta al `StatefulWidget` en el que se define.
- Para cambios de UI simples, como la visibilidad de un elemento, el texto de un contador, o el estado de un checkbox.
- Cuando no necesitas compartir el estado con muchos otros widgets o acceder a él desde muy lejos en el árbol de widgets.

Ejemplo de `setState` (repaso):

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

class ContadorSimple extends StatefulWidget {
  const ContadorSimple({Key? key}) : super(key: key);

  @override
  State<ContadorSimple> createState() => _ContadorSimpleState();
}

class _ContadorSimpleState extends State<ContadorSimple> {
  int _conteo = 0;

  void _incrementar() {
    setState(() {
      _conteo++;
```

```
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Contador Simple')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text(
              'Has presionado el botón esta cantidad de veces:',
              style: TextStyle(fontSize: 18),
            ),
            Text(
              '$_conteo',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementar,
        tooltip: 'Incrementar',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

3.2.2 `InheritedWidget` para el estado compartido

`InheritedWidget` es un tipo especial de widget que permite a sus hijos acceder a los datos que contiene de manera eficiente. Es la forma más fundamental en Flutter para pasar datos hacia abajo en el árbol de widgets sin tener que pasar explícitamente los datos a través de cada constructor intermedio. Es la base de muchos paquetes de gestión de estado populares, como Provider y Riverpod.

Características clave de `InheritedWidget`:

- **Propagación eficiente**: Cuando los datos en un `InheritedWidget` cambian, solo los widgets descendientes que dependen de esos datos se reconstruyen. Esto lo hace muy eficiente.
- **Acceso a datos**: Los widgets descendientes pueden acceder a los datos de un `InheritedWidget` utilizando el método estático `of(BuildContext context)`.

- **Inmutabilidad**: Un `InheritedWidget` en sí mismo es inmutable. Para cambiar los datos, se debe crear una nueva instancia del `InheritedWidget`.

Cuándo usar `InheritedWidget` directamente:

- Para compartir datos que cambian con poca frecuencia o que son relativamente estáticos (por ejemplo, información de usuario, configuraciones de tema).
- Como base para construir soluciones de gestión de estado más complejas.
- Cuando necesitas un control muy fino sobre cuándo se reconstruyen los widgets descendientes.

Ejemplo de `InheritedWidget`:

Imaginemos que queremos compartir un mensaje global a través de nuestra aplicación.

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

class MensajeGlobal extends InheritedWidget {
  const MensajeGlobal({
    Key? key,
    required this.mensaje,
    required Widget child,
  }) : super(key: key, child: child);

  final String mensaje;

  static String? of(BuildContext context) {
    return
context.dependOnInheritedWidgetOfExactType<MensajeGlobal>()?.mensaje;
  }

  @override
  bool updateShouldNotify(MensajeGlobal oldWidget) {
    return mensaje != oldWidget.mensaje;
  }
}

class PaginaPrincipal extends StatelessWidget {
  const PaginaPrincipal({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Página Principal')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text(
```

```
              'Mensaje de la aplicación:',
              style: TextStyle(fontSize: 20),
            ),
            MensajeDisplay(),
            ElevatedButton(
              onPressed: () {
                // Para cambiar el mensaje, necesitaríamos un
StatefulWidget padre
                // que reconstruya MensajeGlobal con un nuevo
mensaje.
                // Esto se gestiona mejor con patrones como
Provider.
              },
              child: const Text('Cambiar Mensaje (no
implementado aquí)'),
            ),
          ],
        ),
      ),
    );
  }
}

class MensajeDisplay extends StatelessWidget {
  const MensajeDisplay({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final mensaje = MensajeGlobal.of(context) ?? 'No hay
mensaje';
    return Text(
      mensaje,
      style: Theme.of(context).textTheme.headlineSmall,
    );
  }
}

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MensajeGlobal(
      mensaje: '¡Bienvenido a mi aplicación Flutter!',
      child: const MaterialApp(
        home: PaginaPrincipal(),
      ),
    );
```

```
  }
}
```

En este ejemplo, `MensajeGlobal` es un `InheritedWidget` que contiene una cadena `mensaje`. El método estático `of(BuildContext context)` permite a cualquier widget descendiente (como `MensajeDisplay`) acceder a este mensaje. El método `updateShouldNotify` es crucial; devuelve `true` si los widgets que dependen de este `InheritedWidget` deben reconstruirse cuando el widget cambia.

Aunque `InheritedWidget` es potente, su uso directo para el estado mutable puede ser un poco engorroso, ya que requiere un `StatefulWidget` padre para reconstruir el `InheritedWidget` con nuevos datos. Es por eso que a menudo se utilizan soluciones de gestión de estado que abstraen esta complejidad, como Provider, que se basan en `InheritedWidget` internamente pero ofrecen una API más sencilla y reactiva.

3.3 Proveedores (Provider, Riverpod)

El paquete `provider` es una de las soluciones de gestión de estado más populares y recomendadas en Flutter. Se basa en `InheritedWidget` pero simplifica enormemente su uso, proporcionando una API más sencilla y reactiva. `Riverpod` es una reimplementación de `provider` que aborda algunas de sus limitaciones, ofreciendo una mayor seguridad de tipo y flexibilidad.

3.3.1 El paquete `provider`

`provider` es una combinación de `InheritedWidget` y `ChangeNotifier` (o cualquier otro tipo de objeto que pueda notificar a sus oyentes sobre cambios). Su filosofía es simple: "proporcionar" un valor a los widgets descendientes en el árbol de widgets y "escuchar" los cambios en ese valor para reconstruir la UI.

Conceptos clave de `provider`:

- **`ChangeNotifier`**: Una clase que puedes extender para crear un objeto que notifica a sus oyentes cuando cambia. Es la forma más común de exponer el estado mutable con `provider`.
- **`ChangeNotifierProvider`**: Un widget que crea una instancia de un `ChangeNotifier` y la proporciona a sus descendientes. Cuando el `ChangeNotifier` notifica un cambio, los widgets que lo "escuchan" se reconstruyen.

- **`Consumer`**: Un widget que "escucha" los cambios en un `ChangeNotifierProvider` y reconstruye solo la parte de la UI que depende de ese estado. Es una forma eficiente de optimizar las reconstrucciones.
- **`Provider.of<T>(context)`**: Un método para acceder a un proveedor. Puedes usar `listen: true` (por defecto) para que el widget se reconstruya cuando el proveedor cambie, o `listen: false` si solo necesitas leer el valor una vez (por ejemplo, en `initState`).
- **`context.watch<T>()`**: Una extensión de `BuildContext` que es una forma concisa de `Provider.of<T>(context, listen: true)`.
- **`context.read<T>()`**: Una extensión de `BuildContext` que es una forma concisa de `Provider.of<T>(context, listen: false)`.

Ejemplo de `provider` con `ChangeNotifier`:

Vamos a rehacer el ejemplo del contador usando `provider`.

1. **Añadir la dependencia**: En tu archivo `pubspec.yaml`: `yaml dependencies: flutter: sdk: flutter provider: ^6.0.5 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.

2. **Crear el `ChangeNotifier` (Modelo)**: ```dart import 'package:flutter/foundation.dart';

   class ContadorModelo extends ChangeNotifier { int _conteo = 0;

   int get conteo => _conteo;

   void incrementar() { _conteo++; notifyListeners(); // Notifica a los oyentes que el estado ha cambiado } } ```

3. **Proporcionar el `ChangeNotifier`**: Envuelve tu `MaterialApp` (o la parte de la aplicación que necesita acceso al estado) con un `ChangeNotifierProvider`. ```dart import 'package:flutter/material.dart'; import 'package:provider/provider.dart'; import 'contador_modelo.dart'; // Tu archivo ContadorModelo

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

   class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return MaterialApp(title: 'Contador con Provider', theme: ThemeData(primarySwatch: Colors.blue,), home: const ContadorPagina(),); } } ```

4. **Consumir el estado**: Utiliza `Consumer` o `context.watch` para acceder al estado y reconstruir la UI. ```dart import 'package:flutter/material.dart'; import 'package:provider/provider.dart'; import 'contador_modelo.dart';

   class ContadorPagina extends StatelessWidget { const ContadorPagina({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return Scaffold(appBar: AppBar(title: const Text('Contador con Provider')), body: Center(child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [const Text('Valor del contador:', style: TextStyle(fontSize: 20),), // Opción 1: Usar Consumer Consumer(builder: (context, contadorModelo, child) { return Text('${contadorModelo.conteo}', style: Theme.of(context).textTheme.headlineMedium,); },), // Opción 2: Usar context.watch (más conciso) // Text(// '${context.watch().conteo}', // style: Theme.of(context).textTheme.headlineMedium, //),],),), floatingActionButton: FloatingActionButton(onPressed: () { // Opción 1: Usar Provider.of con listen: false // Provider.of(context, listen: false).incrementar(); // Opción 2: Usar context.read (más conciso) context.read().incrementar(); }, tooltip: 'Incrementar', child: const Icon(Icons.add),),); } } ```

`provider` es una excelente opción para la mayoría de las aplicaciones Flutter debido a su simplicidad, rendimiento y la forma en que se integra con el sistema de widgets de Flutter.

3.3.2 El paquete `Riverpod`

`Riverpod` es una alternativa a `provider` que resuelve algunas de sus deficiencias, como la seguridad de tipo y la capacidad de anular proveedores. Está diseñado para ser más seguro, más flexible y más fácil de probar. A diferencia de `provider`, `Riverpod` no depende de `BuildContext` para acceder a los proveedores, lo que lo hace más robusto y menos propenso a errores relacionados con el árbol de widgets.

Conceptos clave de `Riverpod`:

- **`Provider`**: En `Riverpod`, un `Provider` es una forma de definir cómo se crea y se accede a un valor. Hay varios tipos de proveedores (`Provider`, `StateProvider`, `StateNotifierProvider`, `FutureProvider`, `StreamProvider`, etc.).
- **`ConsumerWidget` y `ConsumerStatefulWidget`**: Widgets especiales que permiten a sus hijos "escuchar" los cambios en los proveedores de `Riverpod`.

- **`ref`**: Un objeto `ref` se pasa a los constructores de los proveedores y a los métodos `build` de los `ConsumerWidget`s. Se utiliza para leer otros proveedores, escuchar cambios o realizar acciones.
- **`.watch()`**: Se utiliza para "observar" un proveedor y reconstruir el widget cuando su valor cambia.
- **`.read()`**: Se utiliza para "leer" el valor actual de un proveedor sin escuchar cambios.

Ejemplo de `Riverpod` con `StateProvider`:

Vamos a rehacer el ejemplo del contador usando `Riverpod`.

1. **Añadir la dependencia**: En tu archivo `pubspec.yaml`: `yaml dependencies: flutter: sdk: flutter flutter_riverpod: ^2.4.9 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.

2. **Definir el `Provider`**: Define un `StateProvider` global para el contador. ```dart import 'package:flutter_riverpod/flutter_riverpod.dart';

   // Un StateProvider que expone un int. Podemos modificar su estado. final contadorProvider = StateProvider((ref) => 0); ```

3. **Envolver la aplicación con `ProviderScope`**: Todas las aplicaciones que usan `Riverpod` deben estar envueltas en un `ProviderScope`. ```dart import 'package:flutter/material.dart'; import 'package:flutter_riverpod/flutter_riverpod.dart'; import 'contador_provider.dart'; // Tu archivo contador_provider.dart

   void main() { runApp(// ProviderScope es el widget que almacena el estado de todos los proveedores. const ProviderScope(child: MyApp(),),); }

   class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return MaterialApp(title: 'Contador con Riverpod', theme: ThemeData(primarySwatch: Colors.blue,), home: const ContadorPaginaRiverpod(),); } } ```

4. **Consumir el estado**: Utiliza `ConsumerWidget` para acceder al estado y modificarlo. ```dart import 'package:flutter/material.dart'; import 'package:flutter_riverpod/flutter_riverpod.dart'; import 'contador_provider.dart';

   class ContadorPaginaRiverpod extends ConsumerWidget { const ContadorPaginaRiverpod({Key? key}) : super(key: key);

@override Widget build(BuildContext context, WidgetRef ref) { // Observa el estado del contadorProvider. // Cuando el valor cambia, este widget se reconstruye. final conteo = ref.watch(contadorProvider);

```
 return Scaffold(
   appBar: AppBar(title: const Text('Contador con
Riverpod')),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: [
         const Text(
           'Valor del contador:',
           style: TextStyle(fontSize: 20),
         ),
         Text(
           '$conteo',
           style: Theme.of(context).textTheme.headlineMedium,
         ),
       ],
     ),
   ),
   floatingActionButton: FloatingActionButton(
     onPressed: () {
       // Accede al notifier del StateProvider y modifica su
estado.
       ref.read(contadorProvider.notifier).state++;
     },
     tooltip: 'Incrementar',
     child: const Icon(Icons.add),
   ),
 );
```

}}```

`Riverpod` es una opción excelente para proyectos más grandes o cuando se busca una mayor seguridad de tipo y una arquitectura más robusta. Su enfoque en la inmutabilidad y la inyección de dependencias lo hacen muy potente para la gestión de estado compleja.

Cuándo elegir `provider` vs `Riverpod`:

- **`provider`**: Ideal para la mayoría de las aplicaciones, especialmente si estás empezando con la gestión de estado en Flutter. Es más simple de aprender y usar para casos comunes.
- **`Riverpod`**: Recomendado para aplicaciones más grandes y complejas, o si valoras la seguridad de tipo, la capacidad de anular proveedores para pruebas, y una

arquitectura más explícita y robusta. Puede tener una curva de aprendizaje ligeramente más pronunciada al principio.

Ambos paquetes son mantenidos por el mismo autor (Remi Rousselet) y son muy eficientes. La elección dependerá de la complejidad de tu proyecto y tus preferencias personales.

3.4 BLoC/Cubit

BLoC (Business Logic Component) es un patrón de arquitectura popular en Flutter que separa la lógica de negocio de la interfaz de usuario. Fue introducido por Google y se basa en el concepto de "streams" (flujos de datos) para manejar el estado de la aplicación. `Cubit` es una versión simplificada de BLoC, que utiliza funciones en lugar de eventos para emitir estados, haciéndolo más fácil de usar para casos menos complejos.

3.4.1 El patrón BLoC

El patrón BLoC se centra en la idea de que todo en la aplicación debe ser un flujo de eventos y estados. Los eventos son las entradas (acciones del usuario, datos de la red, etc.), y los estados son las salidas (la forma en que la UI debe reaccionar a esos eventos). El BLoC actúa como un intermediario que toma los eventos, procesa la lógica de negocio y emite nuevos estados.

Componentes clave del patrón BLoC:

- **Eventos**: Representan las acciones que ocurren en la aplicación. Son clases inmutables que se envían al BLoC.
- **Estados**: Representan el estado de la UI en un momento dado. Son clases inmutables que el BLoC emite.
- **BLoC**: Una clase que extiende `Bloc<Event, State>`. Contiene la lógica de negocio y mapea los eventos entrantes a los estados salientes.
- **`bloc_builder`**: Un widget que reconstruye la UI en respuesta a los cambios de estado emitidos por un BLoC.
- **`bloc_listener`**: Un widget que ejecuta una función en respuesta a los cambios de estado de un BLoC, pero no reconstruye la UI. Útil para mostrar `SnackBar`s, navegar, etc.
- **`bloc_consumer`**: Una combinación de `bloc_builder` y `bloc_listener`.
- **`bloc_provider`**: Un widget que proporciona una instancia de un BLoC a sus descendientes en el árbol de widgets.

Ejemplo de BLoC con el contador:

1. **Añadir la dependencia**: En tu archivo `pubspec.yaml`: `yaml dependencies: flutter: sdk: flutter flutter_bloc: ^8.1.3 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.

2. **Definir Eventos**: Crea un archivo `contador_eventos.dart`. ```dart part of 'contador_bloc.dart';

   @immutable abstract class ContadorEvento {}

   class IncrementarContador extends ContadorEvento {} class DecrementarContador extends ContadorEvento {} ```

3. **Definir Estados**: Crea un archivo `contador_estados.dart`. ```dart part of 'contador_bloc.dart';

   @immutable abstract class ContadorEstado { final int valor; const ContadorEstado(this.valor); }

   class ContadorInicial extends ContadorEstado { const ContadorInicial(int valor) : super(valor); }

   class ContadorActualizado extends ContadorEstado { const ContadorActualizado(int valor) : super(valor); } ```

4. **Crear el BLoC**: Crea un archivo `contador_bloc.dart`. ```dart import 'package:bloc/bloc.dart'; import 'package:meta/meta.dart';

   part 'contador_eventos.dart'; part 'contador_estados.dart';

   class ContadorBloc extends Bloc { ContadorBloc() : super(const ContadorInicial(0)) { on((event, emit) { emit(ContadorActualizado(state.valor + 1)); }); on((event, emit) { emit(ContadorActualizado(state.valor - 1)); }); } } ```

5. **Integrar en la UI**: Envuelve tu aplicación con `BlocProvider` y usa `BlocBuilder`. ```dart import 'package:flutter/material.dart'; import 'package:flutter_bloc/flutter_bloc.dart'; import 'contador_bloc.dart';

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

   class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return MaterialApp(title: 'Contador con BLoC', theme: ThemeData(primarySwatch: Colors.blue,), home:

BlocProvider(create: (context) => ContadorBloc(), child: const ContadorPaginaBloc(),),); } }

class ContadorPaginaBloc extends StatelessWidget { const ContadorPaginaBloc({Key? key}) : super(key: key);

@override Widget build(BuildContext context) { return Scaffold(appBar: AppBar(title: const Text('Contador con BLoC')), body: Center(child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [const Text('Valor del contador:', style: TextStyle(fontSize: 20),), BlocBuilder(builder: (context, state) { return Text('${state.valor}', style: Theme.of(context).textTheme.headlineMedium,); },),],),), floatingActionButton: Column(mainAxisAlignment: MainAxisAlignment.end, children: [FloatingActionButton(heroTag: 'incrementar', onPressed: () { context.read().add(IncrementarContador()); }, tooltip: 'Incrementar', child: const Icon(Icons.add),), const SizedBox(height: 10), FloatingActionButton(heroTag: 'decrementar', onPressed: () { context.read().add(DecrementarContador()); }, tooltip: 'Decrementar', child: const Icon(Icons.remove),),],),); } } ```

El patrón BLoC es muy potente para aplicaciones grandes y complejas, ya que impone una clara separación de responsabilidades y facilita la prueba de la lógica de negocio.

3.4.2 Cubit

`Cubit` es una alternativa más simple a BLoC, también parte del paquete `flutter_bloc`. A diferencia de BLoC, que utiliza eventos para desencadenar cambios de estado, Cubit utiliza funciones directamente para emitir nuevos estados. Esto lo hace más conciso y fácil de entender para casos de uso más sencillos.

Componentes clave de Cubit:

- **Cubit**: Una clase que extiende `Cubit<State>`. Contiene la lógica de negocio y expone funciones que emiten nuevos estados.
- **`emit()`**: El método utilizado dentro de un Cubit para emitir un nuevo estado.

Ejemplo de Cubit con el contador:

1. **Crear el Cubit**: Crea un archivo `contador_cubit.dart`. ```dart import 'package:bloc/bloc.dart';

   class ContadorCubit extends Cubit { ContadorCubit() : super(0); // Estado inicial es 0

   void incrementar() => emit(state + 1); void decrementar() => emit(state - 1); } ```

2. **Integrar en la UI**: Envuelve tu aplicación con `BlocProvider` y usa `BlocBuilder` (o `BlocConsumer`). ```dart import 'package:flutter/material.dart'; import 'package:flutter_bloc/flutter_bloc.dart'; import 'contador_cubit.dart';

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

   class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return MaterialApp(title: 'Contador con Cubit', theme: ThemeData(primarySwatch: Colors.blue,), home: BlocProvider(create: (context) => ContadorCubit(), child: const ContadorPaginaCubit(),),); } }

   class ContadorPaginaCubit extends StatelessWidget { const ContadorPaginaCubit({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return Scaffold(appBar: AppBar(title: const Text('Contador con Cubit')), body: Center(child: Column(mainAxisAlignment: MainAxisAlignment.center, children: [const Text('Valor del contador:', style: TextStyle(fontSize: 20),), BlocBuilder(builder: (context, state) { return Text('$state', style: Theme.of(context).textTheme.headlineMedium,); },),],),), floatingActionButton: Column(mainAxisAlignment: MainAxisAlignment.end, children: [FloatingActionButton(heroTag: 'incrementar_cubit', onPressed: () { context.read().incrementar(); }, tooltip: 'Incrementar', child: const Icon(Icons.add),), const SizedBox(height: 10), FloatingActionButton(heroTag: 'decrementar_cubit', onPressed: () { context.read().decrementar(); }, tooltip: 'Decrementar', child: const Icon(Icons.remove),),],),); } } ```

Cubit es una excelente opción para manejar el estado en Flutter cuando la lógica de negocio no es excesivamente compleja y no se requiere la granularidad de los eventos. Es más fácil de aprender y escribir que BLoC, pero sigue ofreciendo los beneficios de la separación de responsabilidades y la capacidad de prueba.

Cuándo elegir BLoC vs Cubit:

- **Cubit**: Preferible para casos de uso más simples, donde la lógica de negocio se puede expresar fácilmente con funciones que emiten estados. Es más conciso y tiene una curva de aprendizaje más suave.
- **BLoC**: Ideal para escenarios más complejos donde necesitas una trazabilidad clara de los eventos, una lógica de negocio más elaborada que requiere mapear

múltiples eventos a diferentes estados, o cuando trabajas en equipos grandes donde la explicitud de los eventos es beneficiosa.

Ambos son parte del mismo paquete `flutter_bloc` y se pueden usar juntos en la misma aplicación, eligiendo el que mejor se adapte a la complejidad de cada parte de la lógica de negocio.

3.5 GetX (opcional, si el espacio lo permite)

GetX es un microframework muy popular en Flutter que ofrece una solución completa para la gestión de estado, inyección de dependencias, enrutamiento y mucho más. Se caracteriza por su alto rendimiento, facilidad de uso y la ausencia de `BuildContext` en muchas de sus operaciones, lo que lo hace muy atractivo para muchos desarrolladores. Aunque es una solución "todo en uno", su enfoque puede ser diferente al de otros patrones más tradicionales como Provider o BLoC.

Conceptos clave de GetX:

- **Gestión de estado reactiva**: GetX utiliza un sistema reactivo simple pero potente. Puedes hacer que cualquier variable sea "observable" y actualizar la UI automáticamente cuando su valor cambia, sin necesidad de `ChangeNotifier` o `StreamController`.
  - **`Obx`**: Un widget que reconstruye solo la parte de la UI que depende de las variables observables.
  - **`.obs`**: Un sufijo que se añade a una variable para hacerla observable (por ejemplo, `var count = 0.obs;`).
- **Controladores (`GetxController`)**: Clases que extienden `GetxController` y contienen la lógica de negocio y el estado de tu aplicación. Se gestionan automáticamente por GetX y se pueden acceder desde cualquier parte de la aplicación.
- **Inyección de dependencias**: GetX proporciona un sistema de inyección de dependencias muy sencillo para registrar y recuperar controladores o cualquier otra clase.
- **Gestión de rutas**: Ofrece una API concisa para la navegación sin `BuildContext`.

Ejemplo de GetX con el contador:

Vamos a rehacer el ejemplo del contador usando GetX.

1. **Añadir la dependencia**: En tu archivo `pubspec.yaml`: `yaml dependencies: flutter: sdk: flutter get: ^4.6.5 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.

2. **Crear el Controlador**: Crea un archivo `contador_controller.dart`. ```dart import 'package:get/get.dart';

   class ContadorController extends GetxController { // Hace la variable 'count' observable. Cuando cambia, Obx se reconstruye. var count = 0.obs;

   void incrementar() { count.value++; // Accede al valor con .value }

   void decrementar() { count.value--; } } ```

3. **Integrar en la UI**: Envuelve tu aplicación con `GetMaterialApp` y usa `Obx`. ```dart import 'package:flutter/material.dart'; import 'package:get/get.dart'; import 'contador_controller.dart';

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

   class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return GetMaterialApp(title: 'Contador con GetX', theme: ThemeData(primarySwatch: Colors.blue,), home: const ContadorPaginaGetX(),); } }

   class ContadorPaginaGetX extends StatelessWidget { const ContadorPaginaGetX({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { // Registra y encuentra la instancia del controlador. // Get.put() crea la instancia si no existe. final ContadorController controller = Get.put(ContadorController());

```dart
return Scaffold(
  appBar: AppBar(title: const Text('Contador con GetX')),
  body: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        const Text(
          'Valor del contador:',
          style: TextStyle(fontSize: 20),
        ),
        // Obx reconstruye solo este Text cuando
controller.count cambia.
        Obx(() => Text(
          '${controller.count.value}',
          style: Theme.of(context).textTheme.headlineMedium,
        )),
      ],
    ),
  ),
```

```
    floatingActionButton: Column(
      mainAxisAlignment: MainAxisAlignment.end,
      children: [
        FloatingActionButton(
          heroTag: 'incrementar_getx',
          onPressed: () {
            controller.incrementar();
          },
          tooltip: 'Incrementar',
          child: const Icon(Icons.add),
        ),
        const SizedBox(height: 10),
        FloatingActionButton(
          heroTag: 'decrementar_getx',
          onPressed: () {
            controller.decrementar();
          },
          tooltip: 'Decrementar',
          child: const Icon(Icons.remove),
        ),
      ],
    ),
  );
```

}}```

GetX es conocido por su sintaxis concisa y su enfoque en la productividad. Elimina gran parte del "boilerplate" (código repetitivo) que a veces se asocia con otras soluciones de gestión de estado. Sin embargo, su naturaleza "todo en uno" y su desviación de algunos patrones de Flutter más convencionales pueden ser un punto de discusión en la comunidad.

Cuándo considerar GetX:

- Si buscas una solución rápida y sencilla para la gestión de estado, inyección de dependencias y enrutamiento.
- Si prefieres una sintaxis más concisa y menos código repetitivo.
- Para proyectos pequeños a medianos donde la velocidad de desarrollo es una prioridad.

Consideraciones:

- **Curva de aprendizaje**: Aunque es fácil de usar, entender completamente cómo funciona GetX internamente y sus implicaciones puede llevar tiempo.
- **Opiniones de la comunidad**: GetX ha generado debates en la comunidad de Flutter debido a su enfoque "todo en uno" y a veces por su uso de patrones que

pueden no alinearse con las mejores prácticas de Flutter (como la ausencia de `BuildContext` en el enrutamiento).
- **Flexibilidad**: Aunque es muy potente, su enfoque puede ser menos flexible para arquitecturas muy complejas o personalizadas en comparación con soluciones más modulares como Provider o BLoC.

En resumen, GetX es una herramienta poderosa que puede acelerar el desarrollo de aplicaciones Flutter, especialmente para aquellos que buscan una solución integrada y de alto rendimiento. La elección de GetX, al igual que cualquier otra solución de gestión de estado, debe basarse en las necesidades específicas del proyecto y las preferencias del equipo de desarrollo.

Capítulo 4: Navegación y Enrutamiento

La navegación es un aspecto fundamental de cualquier aplicación móvil, permitiendo a los usuarios moverse entre diferentes pantallas o vistas. Flutter ofrece un sistema de navegación flexible y potente, que ha evolucionado con el tiempo para adaptarse a las necesidades de aplicaciones complejas. Este capítulo explorará los conceptos de navegación en Flutter, desde el enfoque imperativo de `Navigator 1.0` hasta el declarativo de `Navigator 2.0`, y cómo manejar rutas con nombres y el paso de argumentos.

4.1 Navegación básica (Navigator 1.0)

El `Navigator` es un widget que gestiona un stack de `Route`s. Cada `Route` representa una pantalla o página en tu aplicación. Cuando navegas a una nueva pantalla, se "empuja" una nueva ruta al stack. Cuando regresas, se "saca" la ruta actual del stack. Este modelo de navegación es imperativo, lo que significa que le dices al `Navigator` qué hacer (por ejemplo, "empuja esta ruta", "saca la ruta actual").

4.1.1 `MaterialPageRoute` y `CupertinoPageRoute`

Para crear una nueva ruta y empujarla al stack, comúnmente se utilizan `MaterialPageRoute` para aplicaciones con Material Design y `CupertinoPageRoute` para aplicaciones con estilo iOS. Estos `PageRoute`s se encargan de las transiciones de pantalla específicas de cada plataforma.

Métodos comunes de `Navigator`:

- **`Navigator.push(context, route)`**: Empuja una nueva ruta al stack. La nueva pantalla se muestra encima de la actual.

- **`Navigator.pop(context, [result])`**: Saca la ruta actual del stack, regresando a la pantalla anterior. Opcionalmente, puedes pasar un resultado de vuelta a la pantalla anterior.
- **`Navigator.pushReplacement(context, newRoute)`**: Reemplaza la ruta actual con una nueva ruta. La pantalla anterior se elimina del stack, por lo que no se puede volver a ella.
- **`Navigator.pushAndRemoveUntil(context, newRoute, predicate)`**: Empuja una nueva ruta y elimina todas las rutas anteriores hasta que el `predicate` (una función que devuelve `true` o `false`) sea verdadero. Útil para limpiar el stack de navegación, por ejemplo, después de un inicio de sesión exitoso.

Ejemplo de navegación básica:

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

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Navegación Básica',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: const PrimeraPagina(),
    );
  }
}

class PrimeraPagina extends StatelessWidget {
  const PrimeraPagina({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Primera Página')),
      body: Center(
        child: ElevatedButton(
          child: const Text('Ir a la Segunda Página'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => const SegundaPagina(),
              ),
            );
```

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

class SegundaPagina extends StatelessWidget {
  const SegundaPagina({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Segunda Página')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text('Estás en la Segunda Página'),
            ElevatedButton(
              child: const Text('Volver a la Primera Página'),
              onPressed: () {
                Navigator.pop(context);
              },
            ),
            ElevatedButton(
              child: const Text('Ir a la Tercera Página y
reemplazar'),
              onPressed: () {
                Navigator.pushReplacement(
                  context,
                  MaterialPageRoute(
                    builder: (context) => const TerceraPagina(),
                  ),
                );
              },
            ),
          ],
        ),
      ),
    );
  }
}

class TerceraPagina extends StatelessWidget {
  const TerceraPagina({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Tercera Página')),
      body: Center(
```

```
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text('Estás en la Tercera Página'),
            ElevatedButton(
              child: const Text('Volver al inicio (Primera
Página)'),
              onPressed: () {
                Navigator.pushAndRemoveUntil(
                  context,
                  MaterialPageRoute(builder: (context) => const
PrimeraPagina()),
                  (Route<dynamic> route) => false, // Elimina
todas las rutas anteriores
                );
              },
            ),
          ],
        ),
      ),
    );
  }
}
```

Este enfoque es sencillo para aplicaciones con pocas pantallas y una navegación lineal. Sin embargo, a medida que la aplicación crece y la navegación se vuelve más compleja (por ejemplo, con pestañas, navegación anidada, o deep linking), `Navigator 1.0` puede volverse difícil de manejar.

4.2 Navegación declarativa (Navigator 2.0)

`Navigator 2.0` (también conocido como `Router` API) introduce un enfoque declarativo para la navegación. En lugar de empujar y sacar rutas explícitamente, describes el estado deseado de tu stack de navegación, y Flutter se encarga de realizar las transiciones necesarias para llegar a ese estado. Esto es particularmente útil para aplicaciones web, deep linking, y cuando necesitas un control más fino sobre el stack de navegación.

`Navigator 2.0` se basa en los siguientes componentes clave:

- **`Router`**: El widget principal que gestiona el stack de navegación.
- **`RouterDelegate`**: Una clase que define cómo se construye el stack de `Page`s y cómo se manejan los cambios de ruta. Es el corazón de la lógica de navegación declarativa.

- **`RouteInformationParser`**: Una clase que convierte la información de la ruta del sistema (por ejemplo, la URL en la web) en un objeto de datos que tu `RouterDelegate` puede entender.
- **`Page`**: Una abstracción que representa una pantalla en tu aplicación. A diferencia de `Route`, `Page` es inmutable y se utiliza para construir el stack de navegación.

Ventajas de `Navigator 2.0`:

- **Deep Linking**: Facilita la implementación de deep linking, donde una URL específica puede llevar al usuario directamente a una pantalla particular dentro de la aplicación.
- **Navegación Web**: Es fundamental para el desarrollo web con Flutter, ya que permite que la URL del navegador refleje el estado de la aplicación.
- **Control del Stack**: Ofrece un control más granular sobre el stack de navegación, lo que es útil para escenarios complejos como la autenticación o la navegación anidada.
- **Probabilidad**: Al ser declarativo, el estado de navegación es más fácil de probar.

Desafíos de `Navigator 2.0`:

- **Curva de aprendizaje**: Es significativamente más complejo que `Navigator 1.0` y requiere una comprensión más profunda de sus componentes.
- **Boilerplate**: Puede generar más código repetitivo para configuraciones simples.

Ejemplo conceptual de `Navigator 2.0`:

Un ejemplo completo de `Navigator 2.0` es extenso y va más allá del alcance de una sección introductoria. Sin embargo, conceptualmente, implicaría:

1. Definir un `AppRoutePath` que represente los diferentes estados de navegación (por ejemplo, `HomePagePath`, `DetailsPagePath(id)`).
2. Implementar un `AppRouteInformationParser` para convertir la URL en `AppRoutePath`.
3. Implementar un `AppRouterDelegate` que, dado un `AppRoutePath`, construya la lista de `Page`s que conforman el stack de navegación.

```
// Ejemplo simplificado de cómo se vería la estructura
class MyAppRouterDelegate extends RouterDelegate<AppRoutePath>
    with ChangeNotifier,
PopNavigatorRouterDelegateMixin<AppRoutePath> {
  // ... implementación

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

```
    return Navigator(
      key: navigatorKey,
      pages: [
        MaterialPage(child: HomePage()),
        if (_selectedBook != null) MaterialPage(child:
BookDetailsPage(book: _selectedBook!)),
      ],
      onPopPage: (route, result) {
        // ... lógica para manejar el pop
        return route.didPop(result);
      },
    );
  }
  // ...
}

class AppRouteInformationParser extends
RouteInformationParser<AppRoutePath> {
  // ... implementación para parsear URL a AppRoutePath
}

class AppRoutePath { /* ... */ }
```

Para la mayoría de las aplicaciones, especialmente las que no requieren deep linking complejo o un control preciso del historial del navegador, `Navigator 1.0` sigue siendo una opción viable. Sin embargo, para aplicaciones web o aquellas con requisitos de navegación avanzados, `Navigator 2.0` es la herramienta adecuada. A menudo, los desarrolladores optan por paquetes de terceros que abstraen la complejidad de `Navigator 2.0`, como `go_router` o `auto_route`.

4.3 Rutas con nombres y paso de argumentos

Para simplificar la navegación y hacerla más mantenible, Flutter permite definir rutas con nombres. Esto evita tener que construir `MaterialPageRoute`s explícitamente cada vez que navegas a una pantalla. Además, es común necesitar pasar datos o argumentos entre pantallas.

4.3.1 Rutas con nombres (`Navigator.pushNamed`)

Puedes definir un mapa de rutas con nombres en tu `MaterialApp` utilizando la propiedad `routes`.

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

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

```dart
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Rutas con Nombres',
      theme: ThemeData(primarySwatch: Colors.green),
      initialRoute: '/', // La ruta inicial de la aplicación
      routes: {
        '/': (context) => const HomePage(),
        '/details': (context) => const DetailsPage(),
        '/settings': (context) => const SettingsPage(),
      },
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Página de Inicio')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              child: const Text('Ir a Detalles'),
              onPressed: () {
                Navigator.pushNamed(context, '/details');
              },
            ),
            ElevatedButton(
              child: const Text('Ir a Configuración'),
              onPressed: () {
                Navigator.pushNamed(context, '/settings');
              },
            ),
          ],
        ),
      ),
    );
  }
}

class DetailsPage extends StatelessWidget {
  const DetailsPage({Key? key}) : super(key: key);

  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Página de Detalles')),
      body: const Center(
        child: Text('Contenido de la página de detalles'),
      ),
    );
  }
}

class SettingsPage extends StatelessWidget {
  const SettingsPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Página de
Configuración')),
      body: const Center(
        child: Text('Contenido de la página de configuración'),
      ),
    );
  }
}
```

4.3.2 Paso de argumentos

Hay varias formas de pasar argumentos entre pantallas en Flutter:

1. A través del constructor (recomendado para `Navigator 1.0`):

La forma más sencilla y segura de pasar argumentos es a través del constructor de la clase de la pantalla de destino. Esto hace que los argumentos sean explícitos y fuertemente tipados.

```
// Definición de la página de destino
class DetalleProductoPage extends StatelessWidget {
  final String nombreProducto;
  final double precio;

  const DetalleProductoPage({
    Key? key,
    required this.nombreProducto,
    required this.precio,
  }) : super(key: key);

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

```
      appBar: AppBar(title: Text(nombreProducto)),
      body: Center(
        child: Text('El precio de ${nombreProducto} es: $
{precio.toStringAsFixed(2)}€'),
      ),
    );
  }
}

// Navegación desde la página de origen
ElevatedButton(
  child: const Text('Ver Detalles del Producto'),
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => const DetalleProductoPage(
          nombreProducto: 'Camiseta',
          precio: 25.99,
        ),
      ),
    );
  },
),
```

2. Usando `arguments` con rutas con nombres:

Cuando usas rutas con nombres, puedes pasar un objeto `arguments` al método `pushNamed`. Luego, la pantalla de destino puede acceder a estos argumentos usando `ModalRoute.of(context)!.settings.arguments`.

```
// Página de origen
ElevatedButton(
  child: const Text('Ir a Detalles con Argumentos'),
  onPressed: () {
    Navigator.pushNamed(
      context,
      '/productDetails',
      arguments: {
        'nombre': 'Zapatos Deportivos',
        'precio': 79.99,
      },
    );
  },
),

// En MaterialApp routes:
routes: {
  '/': (context) => const HomePage(),
```

```
  '/productDetails': (context) => const
ProductDetailsPageWithArgs(),
},

// Página de destino
class ProductDetailsPageWithArgs extends StatelessWidget {
  const ProductDetailsPageWithArgs({Key? key}) : super(key:
key);

  @override
  Widget build(BuildContext context) {
    final Map<String, dynamic> args =
ModalRoute.of(context)!.settings.arguments as Map<String,
dynamic>;
    final String nombre = args['nombre'];
    final double precio = args['precio'];

    return Scaffold(
      appBar: AppBar(title: Text(nombre)),
      body: Center(
        child: Text('El precio de ${nombre} es: $
{precio.toStringAsFixed(2)}€'),
      ),
    );
  }
}
```

Este método es menos seguro en cuanto a tipos, ya que los argumentos se pasan como un `Object` genérico y deben ser casteados manualmente. Es propenso a errores si no se manejan correctamente los tipos o si los argumentos esperados no están presentes.

4.3.3 Generación de rutas (`onGenerateRoute`)

Para un control más avanzado sobre la creación de rutas con nombres y el manejo de argumentos, puedes usar la propiedad `onGenerateRoute` en `MaterialApp` . Esta función se llama cuando se intenta navegar a una ruta con nombre que no está definida en el mapa `routes` .

Esto te permite parsear la ruta y sus argumentos de forma centralizada y devolver la `Route` adecuada.

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

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```dart
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Generación de Rutas',
      theme: ThemeData(primarySwatch: Colors.purple),
      onGenerateRoute: (settings) {
        // settings.name es la ruta con nombre (ej. '/producto/
123')
        // settings.arguments son los argumentos pasados (si los
hay)

        if (settings.name == '/') {
          return MaterialPageRoute(builder: (context) => const
HomePage());
        } else if (settings.name!.startsWith('/producto/')) {
          final productId = settings.name!.substring('/
producto/'.length);
          return MaterialPageRoute(
            builder: (context) =>
DetalleProductoPage(productId: productId),
          );
        } else if (settings.name == '/usuario') {
          final args = settings.arguments as Map<String,
String>?;
          final userId = args?['id'] ?? 'desconocido';
          return MaterialPageRoute(
            builder: (context) => PerfilUsuarioPage(userId:
userId),
          );
        }
        // Si la ruta no se encuentra, puedes devolver una
página de error
        return MaterialPageRoute(builder: (context) => const
PaginaError());
      },
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Inicio')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              child: const Text('Ver Producto 123'),
```

```
              onPressed: () {
                Navigator.pushNamed(context, '/producto/123');
              },
            ),
            ElevatedButton(
              child: const Text('Ver Perfil de Usuario 456'),
              onPressed: () {
                Navigator.pushNamed(context, '/usuario',
arguments: {'id': '456'});
              },
            ),
          ],
        ),
      ),
    );
  }
}

class DetalleProductoPage extends StatelessWidget {
  final String productId;

  const DetalleProductoPage({Key? key, required
this.productId}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Producto #$productId')),
      body: Center(
        child: Text('Detalles del producto con ID: $productId'),
      ),
    );
  }
}

class PerfilUsuarioPage extends StatelessWidget {
  final String userId;

  const PerfilUsuarioPage({Key? key, required this.userId}) :
super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Perfil de Usuario #$userId')),
      body: Center(
        child: Text('Detalles del perfil del usuario con ID:
$userId'),
      ),
    );
  }
}
```

```dart
class PaginaError extends StatelessWidget {
  const PaginaError({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Error')),
      body: const Center(
        child: Text('Página no encontrada'),
      ),
    );
  }
}
```

`onGenerateRoute` es una herramienta poderosa para manejar rutas dinámicas y centralizar la lógica de enrutamiento, lo que puede ser muy útil en aplicaciones de tamaño mediano a grande. Para aplicaciones aún más complejas, se recomienda el uso de paquetes de enrutamiento de terceros que construyen sobre `Navigator 2.0` y ofrecen una API más amigable.

4.4 Deep Linking

El Deep Linking permite que un enlace (ya sea de una notificación push, un correo electrónico, un sitio web o incluso otra aplicación) abra tu aplicación Flutter en una pantalla específica, en lugar de simplemente abrir la aplicación en su pantalla de inicio. Esto mejora significativamente la experiencia del usuario al llevarlo directamente al contenido relevante.

La implementación de deep linking en Flutter se gestiona principalmente a través de `Navigator 2.0` y la configuración nativa de la plataforma.

4.4.1 Configuración nativa

Para que tu aplicación responda a deep links, primero debes configurar las plataformas nativas (Android e iOS) para que reconozcan los esquemas de URL o los dominios asociados a tu aplicación.

Android:

En Android, se utilizan los "intent filters" en el archivo `AndroidManifest.xml` para declarar qué URLs puede manejar tu aplicación. Puedes configurar:

- **Esquemas personalizados**: Por ejemplo, `miaplicacion://ruta/a/contenido`.

- **Enlaces de aplicación (App Links)**: Asociar un dominio web con tu aplicación, lo que permite que los enlaces `https://tudominio.com/ruta` abran tu aplicación directamente si está instalada.

Ejemplo en `android/app/src/main/AndroidManifest.xml` (dentro de la actividad principal):

```
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:launchMode="singleTop"
    android:theme="@style/LaunchTheme"
    android:configChanges="orientation|keyboardHidden|keyboard|
screenSize|smallestScreenSize|locale|layoutDirection|fontScale|
screenLayout|density|uiMode"
    android:hardwareAccelerated="true"
    android:windowSoftInputMode="adjustResize">
    <!-- ... otros intent-filters ... -->
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category
android:name="android.intent.category.DEFAULT" />
        <category
android:name="android.intent.category.BROWSABLE" />
        <!-- Para esquemas personalizados -->
        <data android:scheme="miaplicacion"
android:host="ruta" />
        <!-- Para App Links (asegúrate de tener un archivo
assetlinks.json en tu dominio) -->
        <data android:scheme="https"
android:host="tudominio.com" />
    </intent-filter>
</activity>
```

Para App Links, también necesitarás un archivo `assetlinks.json` en tu dominio web para verificar la propiedad de la aplicación.

iOS:

En iOS, se utilizan los "URL Schemes" y los "Universal Links".

- **URL Schemes**: Se configuran en el archivo `Info.plist` de tu proyecto iOS.

  Ejemplo en `ios/Runner/Info.plist`:

  `xml <key>CFBundleURLTypes</key> <array> <dict> <key>CFBundleTypeRole</key> <string>Editor</string>`

`<key>CFBundleURLSchemes</key> <array> <string>miaplicacion</string> <!-- Tu esquema personalizado --> </array> </dict> </array>`

- **Universal Links**: Requieren la configuración de un archivo `apple-app-site-association` en tu dominio web y la habilitación de "Associated Domains" en las capacidades de tu proyecto Xcode.

4.4.2 Manejo de Deep Links en Flutter

Una vez que la configuración nativa está lista, tu aplicación Flutter necesita procesar la URL del deep link y navegar a la pantalla correcta. Esto se hace de manera más efectiva con `Navigator 2.0`.

El `RouterDelegate` y el `RouteInformationParser` son los componentes clave para manejar deep links. El `RouteInformationParser` toma la URL entrante y la convierte en un objeto de datos que tu aplicación puede entender. Luego, el `RouterDelegate` utiliza este objeto para construir el stack de `Page`s adecuado.

Uso de paquetes para Deep Linking:

Implementar `Navigator 2.0` y el manejo de deep links desde cero puede ser complejo. Por esta razón, se recomienda encarecidamente el uso de paquetes de enrutamiento de terceros que simplifican este proceso:

- **`go_router`**: Un paquete popular de Google que simplifica la navegación declarativa y el deep linking. Permite definir rutas como un árbol y manejar parámetros de ruta y consultas de URL de manera sencilla.
- **`auto_route`**: Un generador de código que crea un sistema de enrutamiento fuertemente tipado y declarativo, incluyendo soporte para deep linking.

Ejemplo conceptual con `go_router`:

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

// El GoRouter configura todas las rutas para la aplicación
final _router = GoRouter(
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => const HomePage(),
    ),
    GoRoute(
      path: '/details/:id', // Ruta con parámetro
```

```
      builder: (context, state) {
        final id = state.pathParameters['id'];
        return DetailsPage(id: id!); // Pasa el ID a la página
de detalles
      },
    ),
    GoRoute(
      path: '/settings',
      builder: (context, state) => const SettingsPage(),
    ),
  ],
);

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp.router(
      routerConfig: _router, // Usa el router configurado
      title: 'Deep Linking con GoRouter',
      theme: ThemeData(primarySwatch: Colors.teal),
    );
  }
}

class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Inicio')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              child: const Text('Ir a Detalles del Producto
123'),
              onPressed: () {
                context.go('/details/123'); // Navega usando la
ruta con nombre
              },
            ),
            ElevatedButton(
              child: const Text('Ir a Configuración'),
              onPressed: () {
                context.go('/settings');
              },
```

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

class DetailsPage extends StatelessWidget {
  final String id;

  const DetailsPage({Key? key, required this.id}) : super(key:
key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Detalles del ID: $id')),
      body: Center(
        child: Text('Contenido de los detalles para el ID:
$id'),
      ),
    );
  }
}

class SettingsPage extends StatelessWidget {
  const SettingsPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Página de
Configuración')),
      body: const Center(
        child: Text('Contenido de la página de configuración'),
      ),
    );
  }
}
```

Con `go_router`, la definición de rutas es más intuitiva y el manejo de parámetros de ruta se simplifica. Al combinar la configuración nativa con un paquete de enrutamiento declarativo, puedes implementar deep linking de manera efectiva en tus aplicaciones Flutter, proporcionando una experiencia de usuario fluida y moderna.

Capítulo 5: Consumo de APIs y Persistencia de Datos

En el desarrollo de aplicaciones modernas, es casi inevitable interactuar con servicios externos para obtener o enviar datos, y almacenar información localmente para un acceso rápido o para funcionar sin conexión. Este capítulo aborda cómo consumir APIs RESTful en Flutter y diversas estrategias para la persistencia de datos, desde preferencias simples hasta bases de datos locales y la integración con servicios en la nube como Firebase.

5.1 Consumo de APIs REST (http, Dio)

Las APIs REST (Representational State Transfer) son el estándar de facto para la comunicación entre aplicaciones cliente y servidores. Flutter, al ser una plataforma versátil, ofrece varias formas de interactuar con estas APIs.

5.1.1 El paquete `http`

El paquete `http` es el cliente HTTP oficial de Dart y es una opción ligera y fácil de usar para realizar solicitudes HTTP. Es adecuado para la mayoría de los casos de uso, especialmente para solicitudes GET, POST, PUT y DELETE básicas.

Instalación:

Para usar el paquete `http`, añádelo a tu archivo `pubspec.yaml`:

```yaml
dependencies:
  flutter:
    sdk: flutter
  http: ^1.2.1 # Usa la última versión estable
```

Luego, ejecuta `flutter pub get`.

Ejemplo de uso (GET):

Supongamos que queremos obtener una lista de publicaciones de una API de ejemplo (`jsonplaceholder.typicode.com`).

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

// Modelo de datos para una publicación
class Post {
  final int id;
  final String title;
```

```dart
  final String body;
  final int userId;

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

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'],
      title: json['title'],
      body: json['body'],
      userId: json['userId'],
    );
  }
}

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

  if (response.statusCode == 200) {
    // Si la llamada al servidor fue exitosa, parsea el JSON.
    List<dynamic> body = jsonDecode(response.body);
    return body.map((dynamic item) =>
Post.fromJson(item)).toList();
  } else {
    // Si la llamada no fue exitosa, lanza un error.
    throw Exception('Fallo al cargar las publicaciones');
  }
}

// Ejemplo de uso en un StatefulWidget
import 'package:flutter/material.dart';

class PostsPage extends StatefulWidget {
  const PostsPage({Key? key}) : super(key: key);

  @override
  State<PostsPage> createState() => _PostsPageState();
}

class _PostsPageState extends State<PostsPage> {
  late Future<List<Post>> futurePosts;

  @override
  void initState() {
    super.initState();
    futurePosts = fetchPosts();
```

```
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Publicaciones'),
      ),
      body: Center(
        child: FutureBuilder<List<Post>>(
          future: futurePosts,
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return ListView.builder(
                itemCount: snapshot.data!.length,
                itemBuilder: (context, index) {
                  return Card(
                    margin: const EdgeInsets.all(8.0),
                    child: ListTile(
                      title: Text(snapshot.data![index].title),
                      subtitle: Text(snapshot.data!
[index].body),
                    ),
                  );
                },
              );
            } else if (snapshot.hasError) {
              return Text('${snapshot.error}');
            }
            // Por defecto, muestra un spinner de carga.
            return const CircularProgressIndicator();
          },
        ),
      ),
    );
  }
}
```

Ejemplo de uso (POST):

```
Future<Post> createPost(String title, String body) async {
  final response = await http.post(
    Uri.parse('https://jsonplaceholder.typicode.com/posts'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'title': title,
      'body': body,
      'userId': '1',
```

```
    }),
  );

  if (response.statusCode == 201) {
    // Si la creación fue exitosa, parsea el JSON.
    return Post.fromJson(jsonDecode(response.body));
  } else {
    // Si la creación no fue exitosa, lanza un error.
    throw Exception('Fallo al crear la publicación');
  }
}
```

5.1.2 El paquete `Dio`

`Dio` es un cliente HTTP potente para Dart, que ofrece características avanzadas como interceptores, manejo de formularios, carga/descarga de archivos, tiempo de espera, y más. Es una excelente opción para aplicaciones que requieren una gestión más robusta de las solicitudes HTTP.

Instalación:

```
dependencies:
  flutter:
    sdk: flutter
  dio: ^5.4.0 # Usa la última versión estable
```

Luego, ejecuta `flutter pub get`.

Ejemplo de uso (GET con `Dio`):

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

// Reutilizamos el modelo Post definido anteriormente

Future<List<Post>> fetchPostsDio() async {
  final dio = Dio();
  try {
    final response = await dio.get('https://
jsonplaceholder.typicode.com/posts');
    if (response.statusCode == 200) {
      List<dynamic> body = response.data;
      return body.map((dynamic item) =>
Post.fromJson(item)).toList();
    } else {
      throw Exception('Fallo al cargar las publicaciones: $
{response.statusCode}');
    }
```

```
  } on DioException catch (e) {
    // Manejo de errores específicos de Dio
    if (e.response != null) {
      print('Dio error!');
      print('STATUS: ${e.response?.statusCode}');
      print('DATA: ${e.response?.data}');
      print('HEADERS: ${e.response?.headers}');
    } else {
      // Error sin respuesta (ej. problemas de red)
      print('Error sending request!');
      print(e.message);
    }
    throw Exception('Fallo al cargar las publicaciones: $
{e.message}');
  }
}

// El uso en la UI sería similar al ejemplo con http, solo
cambiando la llamada a fetchPostsDio()
```

Ejemplo de uso (POST con `Dio`):

```
Future<Post> createPostDio(String title, String body) async {
  final dio = Dio();
  try {
    final response = await dio.post(
      'https://jsonplaceholder.typicode.com/posts',
      data: {
        'title': title,
        'body': body,
        'userId': 1,
      },
    );

    if (response.statusCode == 201) {
      return Post.fromJson(response.data);
    } else {
      throw Exception('Fallo al crear la publicación: $
{response.statusCode}');
    }
  } on DioException catch (e) {
    throw Exception('Fallo al crear la publicación: $
{e.message}');
  }
}
```

Interceptores en `Dio`:

Una de las características más potentes de `Dio` son los interceptores, que te permiten interceptar y modificar solicitudes o respuestas. Esto es útil para añadir encabezados de autenticación, registrar solicitudes, manejar errores globalmente, etc.

```
 final dio = Dio();
dio.interceptors.add(InterceptorsWrapper(
  onRequest: (options, handler) {
    // Añadir un token de autenticación, por ejemplo
    // options.headers['Authorization'] = 'Bearer your_token';
    print('REQUEST[${options.method}] => PATH: $
{options.path}');
    return handler.next(options); // Continúa con la solicitud
  },
  onResponse: (response, handler) {
    print('RESPONSE[${response.statusCode}] => PATH: $
{response.requestOptions.path}');
    return handler.next(response); // Continúa con la respuesta
  },
  onError: (DioException e, handler) {
    print('ERROR[${e.response?.statusCode}] => PATH: $
{e.requestOptions.path}');
    return handler.next(e); // Continúa con el error
  },
));
```

La elección entre `http` y `Dio` dependerá de la complejidad de tus necesidades de red. Para solicitudes simples, `http` es suficiente. Para aplicaciones con requisitos de red más avanzados, `Dio` ofrece una mayor flexibilidad y características.

5.2 Serialización y deserialización JSON

Cuando interactúas con APIs REST, los datos suelen intercambiarse en formato JSON (JavaScript Object Notation). Para trabajar con estos datos en Dart y Flutter, necesitas serializarlos (convertir objetos Dart a JSON) y deserializarlos (convertir JSON a objetos Dart).

5.2.1 Serialización manual

Para objetos simples o cuando tienes un control total sobre la estructura JSON, puedes realizar la serialización y deserialización manualmente. Esto implica escribir métodos `fromJson` y `toJson` en tus clases de modelo.

Ejemplo (reutilizando el modelo `Post`):

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

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

  // Deserialización: Crea una instancia de Post desde un mapa
JSON
  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'] as int,
      title: json['title'] as String,
      body: json['body'] as String,
      userId: json['userId'] as int,
    );
  }

  // Serialización: Convierte una instancia de Post a un mapa
JSON
  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'title': title,
      'body': body,
      'userId': userId,
    };
  }
}

// Uso:
// Deserializar:
// String jsonString = '{"id": 1, "title": "foo", "body": "bar",
"userId": 1}';
// Map<String, dynamic> jsonMap = jsonDecode(jsonString);
// Post post = Post.fromJson(jsonMap);

// Serializar:
// Post newPost = Post(id: 101, title: 'Nuevo', body:
'Contenido', userId: 5);
// Map<String, dynamic> postMap = newPost.toJson();
// String jsonOutput = jsonEncode(postMap);
```

Este enfoque es viable para modelos pequeños, pero se vuelve tedioso y propenso a errores a medida que los modelos crecen en complejidad o cuando tienes muchos modelos.

5.2.2 Serialización automática con `json_serializable`

Para modelos de datos más complejos o un gran número de modelos, se recomienda encarecidamente el uso de paquetes de generación de código como `json_serializable`. Este paquete genera automáticamente los métodos `fromJson` y `toJson` por ti, reduciendo el código repetitivo y eliminando errores manuales.

Instalación:

Necesitarás añadir las siguientes dependencias a tu `pubspec.yaml`:

```yaml
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^4.8.1 # Usa la última versión estable

dev_dependencies:
  build_runner: ^2.4.8 # Usa la última versión estable
  json_serializable: ^6.7.1 # Usa la última versión estable
```

Luego, ejecuta `flutter pub get`.

Uso:

1. **Anota tu clase de modelo**: Utiliza las anotaciones `@JsonSerializable()` y `factory` para indicar al generador de código cómo procesar tu clase.

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

   part 'post.g.dart'; // Este archivo será generado automáticamente

   @JsonSerializable() class Post { final int id; final String title; final String body; final int userId;

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

   factory Post.fromJson(Map json) => $PostFromJson(json); Map toJson() => $PostToJson(this); } ```

2. **Ejecuta el generador de código**: Abre tu terminal en la raíz de tu proyecto y ejecuta: `bash flutter pub run build_runner build` O para que se ejecute
   ```

automáticamente cada vez que guardes un archivo: `bash flutter pub run build_runner watch`

Esto generará un archivo `post.g.dart` (o el nombre que hayas especificado con `part`) que contendrá la lógica de serialización y deserialización.

`json_serializable` es una herramienta invaluable para proyectos con modelos de datos complejos, ya que automatiza un proceso propenso a errores y mejora la mantenibilidad del código.

5.3 Persistencia local (shared_preferences, sqflite, Hive)

Además de las APIs, las aplicaciones a menudo necesitan almacenar datos localmente en el dispositivo. Flutter ofrece varias opciones para la persistencia de datos, cada una adecuada para diferentes escenarios.

5.3.1 `shared_preferences`

`shared_preferences` es un paquete que permite almacenar datos simples de tipo clave-valor. Es ideal para guardar configuraciones de usuario, preferencias, o pequeños fragmentos de datos que no requieren una estructura compleja o consultas avanzadas. Internamente, utiliza `NSUserDefaults` en iOS y `SharedPreferences` en Android.

Instalación:

```
dependencies:
  flutter:
    sdk: flutter
  shared_preferences: ^2.2.2 # Usa la última versión estable
```

Luego, ejecuta `flutter pub get`.

Ejemplo de uso:

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

class PreferenciasUsuario {
  static late SharedPreferences _preferences;

  static Future init() async {
    _preferences = await SharedPreferences.getInstance();
  }

  // Guardar un string
  static Future setNombre(String nombre) async {
```

```
    await _preferences.setString('nombre', nombre);
  }

  // Obtener un string
  static String? getNombre() {
    return _preferences.getString('nombre');
  }

  // Guardar un booleano
  static Future setModoOscuro(bool isDarkMode) async {
    await _preferences.setBool('modoOscuro', isDarkMode);
  }

  // Obtener un booleano
  static bool getModoOscuro() {
    return _preferences.getBool('modoOscuro') ??
false; // Valor por defecto
  }

  // Eliminar una preferencia
  static Future removeNombre() async {
    await _preferences.remove('nombre');
  }
}

// Uso en la aplicación (por ejemplo, en main.dart o en
initState de un widget)
void main() async {
  WidgetsFlutterBinding.ensureInitialized(); // Asegura que
Flutter esté inicializado
  await PreferenciasUsuario.init(); // Inicializa
SharedPreferences
  runApp(const MyApp());
}

// En un widget:
// String? nombre = PreferenciasUsuario.getNombre();
// PreferenciasUsuario.setNombre('Juan');
```

5.3.2 `sqflite`

`sqflite` es un plugin de Flutter para SQLite, una base de datos relacional ligera y embebida. Es ideal para almacenar datos estructurados que requieren consultas complejas, relaciones entre tablas o un gran volumen de información. Es la opción preferida para bases de datos locales en aplicaciones móviles.

Instalación:

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.3.0 # Usa la última versión estable
  path_provider: ^2.1.1 # Necesario para obtener la ruta de la
base de datos
  path: ^1.8.3 # Necesario para unir rutas
```

Luego, ejecuta `flutter pub get`.

Ejemplo de uso (CRUD simple):

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class Todo {
  final int? id;
  final String title;
  final String description;
  final bool isDone;

  Todo({
    this.id,
    required this.title,
    required this.description,
    this.isDone = false,
  });

  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'title': title,
      'description': description,
      'isDone': isDone ? 1 : 0,
    };
  }

  factory Todo.fromMap(Map<String, dynamic> map) {
    return Todo(
      id: map['id'],
      title: map['title'],
      description: map['description'],
      isDone: map['isDone'] == 1,
    );
  }

  @override
  String toString() {
    return 'Todo{id: $id, title: $title, description:
```

```
$description, isDone: $isDone}';
  }
}

class DatabaseHelper {
  static Database? _database;
  static const String tableName = 'todos';

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDB();
    return _database!;
  }

  Future<Database> _initDB() async {
    String path = join(await getDatabasesPath(),
'todos_database.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: (db, version) async {
        await db.execute(
          'CREATE TABLE $tableName(id INTEGER PRIMARY KEY
AUTOINCREMENT, title TEXT, description TEXT, isDone INTEGER)',
        );
      },
    );
  }

  Future<int> insertTodo(Todo todo) async {
    final db = await database;
    return await db.insert(tableName, todo.toMap(),
conflictAlgorithm: ConflictAlgorithm.replace);
  }

  Future<List<Todo>> getTodos() async {
    final db = await database;
    final List<Map<String, dynamic>> maps = await
db.query(tableName);
    return List.generate(maps.length, (i) {
      return Todo.fromMap(maps[i]);
    });
  }

  Future<int> updateTodo(Todo todo) async {
    final db = await database;
    return await db.update(
      tableName,
      todo.toMap(),
      where: 'id = ?',
      whereArgs: [todo.id],
    );
```

```
  }

  Future<int> deleteTodo(int id) async {
    final db = await database;
    return await db.delete(
      tableName,
      where: 'id = ?',
      whereArgs: [id],
    );
  }
}

// Uso en la aplicación:
// final dbHelper = DatabaseHelper();
// await dbHelper.insertTodo(Todo(title: 'Comprar leche',
description: 'En el supermercado'));
// List<Todo> todos = await dbHelper.getTodos();
```

5.3.3 `Hive`

`Hive` es una base de datos NoSQL ligera y rápida para Flutter y Dart. Es una excelente alternativa a `shared_preferences` y `sqflite` cuando necesitas almacenar datos más complejos que clave-valor, pero no quieres la sobrecarga de una base de datos relacional. Es muy rápido y fácil de usar.

Instalación:

```
dependencies:
  flutter:
    sdk: flutter
  hive: ^2.2.3 # Usa la última versión estable
  hive_flutter: ^1.1.0 # Para integración con Flutter

dev_dependencies:
  hive_generator: ^2.0.1 # Generador de código para Hive
  build_runner: ^2.4.8 # Necesario para ejecutar el generador
```

Luego, ejecuta `flutter pub get`.

Uso:

1. **Definir el modelo y generar adaptadores**: Crea tu clase de modelo y anótala con `@HiveType()`. Define los campos con `@HiveField()`.

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

   part 'tarea.g.dart'; // Este archivo será generado automáticamente
   ```

@HiveType(typeId: 0) // typeId debe ser único para cada clase class Tarea extends HiveObject { @HiveField(0) String titulo;

@HiveField(1) String descripcion;

@HiveField(2) bool completada;

Tarea({ required this.titulo, required this.descripcion, this.completada = false, }); } ```

2. **Ejecuta el generador de código**: Abre tu terminal en la raíz de tu proyecto y ejecuta: `bash flutter pub run build_runner build` O para que se ejecute automáticamente cada vez que guardes un archivo: `bash flutter pub run build_runner watch`

   Esto generará un archivo `tarea.g.dart` que contendrá el adaptador de tipo para `Tarea`.

3. **Inicializar Hive y abrir una caja (Box)**:

   ```dart import 'package:hive_flutter/hive_flutter.dart'; import 'tarea.g.dart'; // Importa el archivo generado

   void main() async { WidgetsFlutterBinding.ensureInitialized(); await Hive.initFlutter(); Hive.registerAdapter(TareaAdapter()); // Registra el adaptador generado await Hive.openBox('tareasBox'); // Abre una caja para tus tareas runApp(const MyApp()); } ```

4. **Realizar operaciones CRUD:**

   ```dart // Obtener la caja final tareasBox = Hive.box('tareasBox');

   // Añadir una tarea tareasBox.add(Tarea(titulo: 'Estudiar Flutter', descripcion: 'Capítulo 5'));

   // Obtener todas las tareas List todasLasTareas = tareasBox.values.toList();

   // Actualizar una tarea (usando el índice o la clave) if (todasLasTareas.isNotEmpty) { final primeraTarea = todasLasTareas.first; primeraTarea.completada = true; primeraTarea.save(); // Guarda los cambios en la base de datos }

   // Eliminar una tarea (usando el índice o la clave) if (tareasBox.isNotEmpty) { tareasBox.deleteAt(0); // Elimina la primera tarea } ```

`Hive` es una excelente opción para almacenar datos no relacionales de forma rápida y sencilla, especialmente cuando no necesitas la complejidad de SQL o la sincronización en la nube.

## 5.4 Integración con Firebase (Firestore, Authentication)

Firebase es una plataforma de desarrollo de aplicaciones móviles y web de Google que ofrece una suite de servicios backend, incluyendo bases de datos en la nube (Firestore), autenticación, almacenamiento de archivos, funciones serverless y más. La integración de Firebase con Flutter es muy sencilla y potente.

### 5.4.1 Configuración de Firebase en tu proyecto Flutter

1. **Crear un proyecto Firebase**: Ve a la consola de Firebase (console.firebase.google.com) y crea un nuevo proyecto.
2. **Añadir aplicaciones a Firebase**: Sigue las instrucciones en la consola para añadir una aplicación Android y/o iOS a tu proyecto Firebase. Esto implicará descargar archivos de configuración ( `google-services.json` para Android y `GoogleService-Info.plist` para iOS) y añadirlos a tu proyecto Flutter.

3. **Añadir dependencias de FlutterFire**: FlutterFire es la colección oficial de plugins de Firebase para Flutter. Añade las dependencias necesarias a tu `pubspec.yaml` :

   `yaml dependencies: flutter: sdk: flutter firebase_core: ^2.24.2 # Inicialización de Firebase firebase_auth: ^4.15.3 # Autenticación cloud_firestore: ^4.13.5 # Base de datos Firestore`

   Luego, ejecuta `flutter pub get` .

4. **Inicializar Firebase en tu aplicación**: Asegúrate de inicializar Firebase antes de usar cualquier servicio.

   ```dart import 'package:firebase_core/firebase_core.dart'; import 'package:flutter/material.dart'; // Importa tu archivo firebase_options.dart generado por flutterfire_cli import 'firebase_options.dart';

   void main() async { WidgetsFlutterBinding.ensureInitialized(); await Firebase.initializeApp( options: DefaultFirebaseOptions.currentPlatform, ); runApp(const MyApp()); } ```

   Para generar `firebase_options.dart` , puedes usar `flutterfire_cli` : `bash dart pub global activate flutterfire_cli flutterfire configure`
   ```

5.4.2 Firebase Authentication

Firebase Authentication proporciona servicios de backend, SDKs fáciles de usar y bibliotecas de UI listas para usar para autenticar usuarios en tu aplicación. Soporta autenticación con contraseñas, números de teléfono, proveedores de identidad populares como Google, Facebook, Twitter, etc.

Ejemplo de autenticación con correo y contraseña:

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

class AuthService {
  final FirebaseAuth _auth = FirebaseAuth.instance;

  // Registrar usuario con correo y contraseña
  Future<User?> registerWithEmailAndPassword(String email,
String password) async {
    try {
      UserCredential result = await
_auth.createUserWithEmailAndPassword(email: email, password:
password);
      return result.user;
    } catch (e) {
      print(e.toString());
      return null;
    }
  }

  // Iniciar sesión con correo y contraseña
  Future<User?> signInWithEmailAndPassword(String email, String
password) async {
    try {
      UserCredential result = await
_auth.signInWithEmailAndPassword(email: email, password:
password);
      return result.user;
    } catch (e) {
      print(e.toString());
      return null;
    }
  }

  // Cerrar sesión
  Future<void> signOut() async {
    await _auth.signOut();
  }

  // Obtener el usuario actual (Stream para cambios de estado de
autenticación)
  Stream<User?> get user => _auth.authStateChanges();
```

```
}

// Uso en la UI (ejemplo simplificado)
// final _authService = AuthService();
// User? user = await
_authService.signInWithEmailAndPassword('test@example.com',
'password123');
// _authService.user.listen((user) { /* Manejar cambios de
estado de autenticación */ });
```

5.4.3 Cloud Firestore

Cloud Firestore es una base de datos NoSQL flexible y escalable para el desarrollo móvil, web y de servidores. Almacena datos en documentos organizados en colecciones. Ofrece sincronización en tiempo real y soporte offline, lo que la hace ideal para aplicaciones que necesitan datos actualizados al instante.

Ejemplo de operaciones CRUD con Firestore:

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

class TodoFirestoreService {
  final FirebaseFirestore _firestore =
FirebaseFirestore.instance;

  // Añadir una nueva tarea
  Future<void> addTodo(String title, String description) async {
    await _firestore.collection('todos').add({
      'title': title,
      'description': description,
      'isDone': false,
      'timestamp': FieldValue.serverTimestamp(), // Marca de
tiempo del servidor
    });
  }

  // Obtener todas las tareas (en tiempo real)
  Stream<List<Todo>> getTodos() {
    return _firestore.collection('todos').orderBy('timestamp',
descending: true).snapshots().map((snapshot) {
      return snapshot.docs.map((doc) {
        return Todo(
          id: doc.id, // El ID del documento es la clave de
Firestore
          title: doc['title'],
          description: doc['description'],
          isDone: doc['isDone'],
        );
      }).toList();
```

```dart
    });
  }

  // Actualizar una tarea
  Future<void> updateTodo(String id, bool isDone) async {
    await _firestore.collection('todos').doc(id).update({
      'isDone': isDone,
    });
  }

  // Eliminar una tarea
  Future<void> deleteTodo(String id) async {
    await _firestore.collection('todos').doc(id).delete();
  }
}

// Reutilizamos el modelo Todo, pero el id ahora es String
class Todo {
  final String? id;
  final String title;
  final String description;
  final bool isDone;

  Todo({
    this.id,
    required this.title,
    required this.description,
    this.isDone = false,
  });
}

// Uso en la UI (ejemplo simplificado con StreamBuilder)
// final _todoService = TodoFirestoreService();
// StreamBuilder<List<Todo>>(
//   stream: _todoService.getTodos(),
//   builder: (context, snapshot) {
//     if (snapshot.hasData) {
//       return ListView.builder(
//         itemCount: snapshot.data!.length,
//         itemBuilder: (context, index) {
//           final todo = snapshot.data![index];
//           return ListTile(
//             title: Text(todo.title),
//             trailing: Checkbox(
//               value: todo.isDone,
//               onChanged: (bool? value) {
//                 _todoService.updateTodo(todo.id!, value!); //
Actualizar en Firestore
//               },
//             ),
//           );
//         },
```

```
//       );
//     } else if (snapshot.hasError) {
//       return Text('Error: ${snapshot.error}');
//     }
//     return CircularProgressIndicator();
//   },
// );
```

La integración con Firebase simplifica enormemente el desarrollo de aplicaciones que requieren backend, permitiendo a los desarrolladores centrarse en la lógica del frontend y la experiencia del usuario. Es una solución escalable y rentable para muchos tipos de aplicaciones.

Capítulo 6: Pruebas en Flutter

Las pruebas son una parte esencial del ciclo de vida del desarrollo de software, y el desarrollo de aplicaciones Flutter no es una excepción. Escribir pruebas ayuda a asegurar la calidad del código, prevenir regresiones, facilitar el refactoring y mejorar la confianza en la aplicación. Flutter proporciona un marco de pruebas robusto que permite escribir diferentes tipos de pruebas para cubrir todos los aspectos de tu aplicación.

6.1 Tipos de pruebas (unitarias, de widgets, de integración)

Flutter distingue principalmente tres tipos de pruebas, cada una con un propósito y alcance diferentes:

6.1.1 Pruebas Unitarias (Unit Tests)

Las pruebas unitarias verifican una sola unidad de código (una función, un método, una clase) de forma aislada, sin dependencias externas como la UI, la base de datos o la red. Su objetivo es asegurar que cada unidad de código funciona como se espera.

Características:

- **Aislamiento**: Se centran en una pequeña parte del código.
- **Velocidad**: Son muy rápidas de ejecutar, lo que permite ejecutarlas con frecuencia.
- **Cobertura**: Ayudan a asegurar que la lógica de negocio subyacente es correcta.

Cuándo usarlas:

- Para probar funciones puras.
- Para verificar la lógica de negocio en clases que no son widgets (por ejemplo, servicios, modelos, utilidades).

- Para asegurar que los algoritmos funcionan correctamente.

6.1.2 Pruebas de Widgets (Widget Tests)

Las pruebas de widgets (también conocidas como pruebas de componentes o pruebas de UI) verifican que un solo widget o un pequeño subárbol de widgets se renderiza y se comporta como se espera. Estas pruebas se ejecutan en un entorno de prueba especial que simula una parte del árbol de widgets de Flutter, permitiendo interactuar con los widgets y verificar su estado y apariencia.

Características:

- **Simulación de UI**: Permiten "montar" widgets en un entorno de prueba.
- **Interacción**: Puedes simular gestos de usuario (taps, scrolls) y entradas de texto.
- **Verificación de UI**: Permiten verificar la apariencia y el estado de los widgets, incluyendo textos, colores, visibilidad de elementos, etc.

Cuándo usarlas:

- Para verificar que los widgets se construyen correctamente con los datos proporcionados.
- Para probar la interacción del usuario con los widgets (por ejemplo, un botón que cambia de texto al ser presionado).
- Para asegurar que los widgets responden adecuadamente a los cambios de estado.

6.1.3 Pruebas de Integración (Integration Tests)

Las pruebas de integración verifican que diferentes partes de tu aplicación (por ejemplo, múltiples widgets, servicios, bases de datos) funcionan correctamente juntas. Se ejecutan en un dispositivo real o en un emulador/simulador, lo que permite probar la aplicación en un entorno lo más cercano posible al de producción.

Características:

- **Entorno real**: Se ejecutan en un dispositivo o emulador real.
- **Flujos de usuario completos**: Permiten probar flujos de usuario complejos que involucran múltiples pantallas y componentes.
- **Rendimiento**: Son más lentas que las pruebas unitarias y de widgets.

Cuándo usarlas:

- Para probar la navegación entre pantallas.
- Para verificar la interacción de la aplicación con servicios externos (APIs, bases de datos).
- Para asegurar que los flujos de usuario críticos funcionan de principio a fin.

La combinación de estos tres tipos de pruebas proporciona una cobertura integral para tu aplicación Flutter, desde la lógica más granular hasta los flujos de usuario completos.

6.2 Escritura de pruebas unitarias

Para escribir pruebas unitarias en Flutter, se utiliza el paquete `test` de Dart. Este paquete proporciona las funciones y clases necesarias para definir y ejecutar pruebas.

Configuración:

Para usar el paquete `test`, añádelo a la sección `dev_dependencies` de tu `pubspec.yaml`:

```yaml
dev_dependencies:
  flutter_test:
    sdk: flutter
  test: ^1.24.9 # Asegúrate de tener la última versión
```

Luego, ejecuta `flutter pub get`.

Estructura de un archivo de prueba unitaria:

Las pruebas unitarias se suelen colocar en el directorio `test/`. Por ejemplo, si tienes una clase `calculadora.dart` en `lib/`, su prueba unitaria podría estar en `test/calculadora_test.dart`.

```dart
// lib/calculadora.dart
class Calculadora {
  double sumar(double a, double b) {
    return a + b;
  }

  double restar(double a, double b) {
    return a - b;
  }

  double multiplicar(double a, double b) {
    return a * b;
  }

  double dividir(double a, double b) {
    if (b == 0) {
      throw ArgumentError("No se puede dividir por cero");
    }
    return a / b;
  }
}
```

```dart
// test/calculadora_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:tu_app/
calculadora.dart'; // Asegúrate de que la ruta sea correcta

void main() {
  group('Calculadora', () {
    late Calculadora calculadora; // Usamos late para
inicializar en setUp

    setUp(() {
      // Se ejecuta antes de cada prueba en este grupo
      calculadora = Calculadora();
    });

    test('sumar debe retornar la suma de dos números', () {
      expect(calculadora.sumar(2, 3), 5);
      expect(calculadora.sumar(-1, 1), 0);
      expect(calculadora.sumar(0, 0), 0);
    });

    test('restar debe retornar la resta de dos números', () {
      expect(calculadora.restar(5, 2), 3);
      expect(calculadora.restar(2, 5), -3);
    });

    test('multiplicar debe retornar el producto de dos
números', () {
      expect(calculadora.multiplicar(2, 3), 6);
      expect(calculadora.multiplicar(5, 0), 0);
    });

    test('dividir debe retornar el cociente de dos números', ()
{
      expect(calculadora.dividir(6, 3), 2);
      expect(calculadora.dividir(10, 2), 5);
    });

    test('dividir debe lanzar un ArgumentError si el divisor es
cero', () {
      expect(() => calculadora.dividir(10, 0),
throwsA(isA<ArgumentError>()));
    });
  });
}
```

Explicación:

- **`void main()`**: Es el punto de entrada para las pruebas. Dentro de `main`, puedes definir tus pruebas.

- **`group()`**: Se utiliza para agrupar pruebas relacionadas. Esto ayuda a organizar las pruebas y a generar informes más legibles.
- **`setUp()`**: Una función que se ejecuta antes de cada prueba dentro de su `group`. Es útil para inicializar objetos o configurar el entorno de prueba.
- **`test()`**: Define una prueba individual. Toma una descripción de la prueba y una función que contiene la lógica de la prueba.
- **`expect()`**: La función principal para hacer aserciones. Compara un valor real con un valor esperado. Si la aserción falla, la prueba falla.
- **`throwsA(isA<TipoDeError>())`**: Se utiliza para verificar que una función lanza una excepción específica.

Ejecución de pruebas unitarias:

Para ejecutar todas las pruebas en tu proyecto, abre una terminal en la raíz de tu proyecto y ejecuta:

```
flutter test
```

Para ejecutar un archivo de prueba específico:

```
flutter test test/calculadora_test.dart
```

Para ejecutar una prueba específica dentro de un archivo (puedes usar el nombre de la prueba o una expresión regular):

```
flutter test test/calculadora_test.dart --name "sumar"
```

Las pruebas unitarias son la base de una estrategia de pruebas sólida y deben ser el primer tipo de prueba que escribas para tu lógica de negocio.

6.3 Escritura de pruebas de widgets

Las pruebas de widgets te permiten verificar la apariencia y el comportamiento de tus widgets de Flutter. Para esto, Flutter proporciona el paquete `flutter_test` y la clase `WidgetTester`.

Configuración:

El paquete `flutter_test` ya está incluido por defecto en la sección `dev_dependencies` de tu `pubspec.yaml` cuando creas un nuevo proyecto Flutter.

Estructura de un archivo de prueba de widgets:

Las pruebas de widgets también se colocan en el directorio `test/`. Por ejemplo, si tienes un widget `mi_boton.dart` en `lib/widgets/`, su prueba podría estar en `test/widgets/mi_boton_test.dart`.

```
// lib/widgets/mi_boton.dart
import 'package:flutter/material.dart';

class MiBoton extends StatelessWidget {
  final String texto;
  final VoidCallback onPressed;

  const MiBoton({
    Key? key,
    required this.texto,
    required this.onPressed,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(texto),
    );
  }
}

// test/widgets/mi_boton_test.dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:tu_app/widgets/mi_boton.dart'; // Asegúrate de
que la ruta sea correcta

void main() {
  group('MiBoton', () {
    testWidgets('MiBoton muestra el texto correcto',
(WidgetTester tester) async {
      // Construye nuestro widget y dispara un frame.
      await tester.pumpWidget(MaterialApp(home: MiBoton(texto:
'Haz Clic', onPressed: () {})));

      // Busca el texto 'Haz Clic' en el árbol de widgets.
      expect(find.text('Haz Clic'), findsOneWidget);

      // Verifica que no haya otro texto.
      expect(find.text('Otro Texto'), findsNothing);
    });

    testWidgets('MiBoton llama a onPressed cuando se presiona',
```

```
(WidgetTester tester) async {
      bool botonPresionado = false;

      // Construye nuestro widget con un callback que cambia una
variable.
      await tester.pumpWidget(MaterialApp(
        home: MiBoton(
          texto: 'Presioname',
          onPressed: () {
            botonPresionado = true;
          },
        ),
      ));

      // Busca el botón por su texto y simula un tap.
      await tester.tap(find.text('Presioname'));

      // Reconstruye el widget después del tap para que los
cambios se reflejen.
      await tester.pump();

      // Verifica que el callback fue llamado.
      expect(botonPresionado, isTrue);
    });
  });
}
```

Explicación:

- **`testWidgets()`**: Define una prueba de widget. Toma una descripción y una función que recibe un `WidgetTester`.
- **`WidgetTester tester`**: Es la herramienta principal para interactuar con los widgets en el entorno de prueba. Permite:
  - **`tester.pumpWidget(widget)`**: Construye el widget que quieres probar y lo añade al árbol de widgets de prueba. También dispara un frame para que el widget se renderice.
  - **`tester.pump()`**: Dispara un frame para que los cambios en el widget se rendericen. Útil después de una interacción o un cambio de estado.
  - **`tester.tap(finder)`**: Simula un tap en el widget encontrado por el `finder`.
  - **`tester.enterText(finder, text)`**: Simula la entrada de texto en un `TextField`.
  - **`tester.scroll(finder, offset)`**: Simula un scroll.

- **`find`**: Un conjunto de funciones para encontrar widgets en el árbol de widgets de prueba. Algunos `finders` comunes son:
  - **`find.text('texto')`**: Encuentra un widget que muestra el texto especificado.
  - **`find.byType(TipoDeWidget)`**: Encuentra widgets de un tipo específico.
  - **`find.byKey(Key)`**: Encuentra un widget por su `Key` (útil para identificar widgets únicos).
  - **`find.byIcon(IconData)`**: Encuentra un widget que muestra un icono específico.
- **`findsOneWidget`**: Un `Matcher` que verifica que se encontró exactamente un widget.
- **`findsNothing`**: Un `Matcher` que verifica que no se encontró ningún widget.
- **`findsNWidgets(n)`**: Un `Matcher` que verifica que se encontraron `n` widgets.
- **`findsWidgets`**: Un `Matcher` que verifica que se encontró al menos un widget.

Ejecución de pruebas de widgets:

Las pruebas de widgets se ejecutan de la misma manera que las pruebas unitarias:

```
 flutter test
flutter test test/widgets/mi_boton_test.dart
```

Las pruebas de widgets son cruciales para asegurar que tu interfaz de usuario se comporta como se espera y que los usuarios tendrán una experiencia consistente y libre de errores.

6.4 Escritura de pruebas de integración

Las pruebas de integración verifican el comportamiento de tu aplicación en un entorno real o simulado, abarcando múltiples widgets y la interacción con servicios externos. Flutter proporciona el paquete `integration_test` para escribir estas pruebas.

Configuración:

1. **Añadir la dependencia**: En tu `pubspec.yaml`: `yaml dev_dependencies: flutter_test: sdk: flutter integration_test: ^2.0.1 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.

2. **Crear el directorio `integration_test`**: Crea un nuevo directorio llamado `integration_test` en la raíz de tu proyecto (al mismo nivel que `lib` y `test`).

3. **Crear el archivo de prueba**: Dentro de `integration_test`, crea un archivo para tus pruebas de integración, por ejemplo, `app_test.dart`.

Estructura de un archivo de prueba de integración:

Un archivo de prueba de integración tiene dos partes principales: la función `main()` que inicializa el entorno de prueba y ejecuta la aplicación, y las pruebas en sí.

```dart
// integration_test/app_test.dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';

import 'package:tu_app/main.dart' as app; // Importa tu archivo main.dart

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('End-to-end Test', () {
    testWidgets('Verificar flujo de inicio de sesión y contador', (WidgetTester tester) async {
      app.main(); // Ejecuta la función main de tu aplicación
      await tester.pumpAndSettle(); // Espera a que la aplicación se estabilice

      // Simular inicio de sesión (si tu app tiene uno)
      // await tester.enterText(find.byType(TextField).first, 'usuario@example.com');
      // await tester.enterText(find.byType(TextField).last, 'password');
      // await tester.tap(find.byType(ElevatedButton));
      // await tester.pumpAndSettle();

      // Verificar que estamos en la página principal (por ejemplo, buscando un texto)
      expect(find.text('Contador: 0'), findsOneWidget);

      // Simular un tap en el botón de incrementar
      await tester.tap(find.byIcon(Icons.add));
      await tester.pump(); // Reconstruye el widget después del tap

      // Verificar que el contador se ha incrementado
      expect(find.text('Contador: 1'), findsOneWidget);

      // Simular otro tap
      await tester.tap(find.byIcon(Icons.add));
      await tester.pump();
```

```
      // Verificar el nuevo valor
      expect(find.text('Contador: 2'), findsOneWidget);
    });

    testWidgets('Navegar a otra página y volver', (WidgetTester
tester) async {
      app.main();
      await tester.pumpAndSettle();

      // Asumiendo que tienes un botón para navegar a una
segunda página
      // await tester.tap(find.text('Ir a Segunda Página'));
      // await tester.pumpAndSettle();

      // expect(find.text('Segunda Página'), findsOneWidget);

      // await tester.pageBack(); // Simula el botón de
retroceso del sistema
      // await tester.pumpAndSettle();

      // expect(find.text('Primera Página'), findsOneWidget);
    });
  });
}
```

Explicación:

- **`IntegrationTestWidgetsFlutterBinding.ensureInitialized()`**: Debe llamarse al principio de la función `main()` de tu archivo de prueba de integración para inicializar el binding de pruebas de integración.
- **`app.main()`**: Llama a la función `main()` de tu aplicación para iniciarla en el entorno de prueba.
- **`tester.pumpAndSettle()`**: Es una función muy importante en las pruebas de integración. Espera a que todas las animaciones y microtareas pendientes se completen. Úsala después de cualquier acción que pueda desencadenar cambios asíncronos en la UI (como navegación, llamadas a API, etc.).
- Los `finders` y `matchers` son los mismos que en las pruebas de widgets.

Ejecución de pruebas de integración:

Para ejecutar pruebas de integración, necesitas un dispositivo o emulador conectado y funcionando. Luego, ejecuta el siguiente comando desde la raíz de tu proyecto:

```
flutter test integration_test/
```

Esto ejecutará todas las pruebas en el directorio `integration_test/`. También puedes especificar un archivo de prueba específico.

Generación de informes:

El paquete `integration_test` puede generar informes en formato JSON o JUnit XML, lo que es útil para la integración continua (CI).

```
flutter test integration_test/app_test.dart --coverage --test-
randomize-ordering-seed=random --reporter json >
integration_test_report.json
```

Las pruebas de integración son vitales para validar los flujos de usuario críticos y asegurar que la aplicación funciona correctamente en un entorno de ejecución realista.

6.5 Mocking y simulación

En las pruebas, especialmente en las unitarias y de widgets, a menudo necesitas aislar la unidad de código que estás probando de sus dependencias externas (como APIs de red, bases de datos, o servicios de autenticación). Aquí es donde entran en juego el mocking y la simulación. El mocking implica crear objetos falsos (mocks) que imitan el comportamiento de las dependencias reales, permitiéndote controlar su respuesta y verificar que tu código interactúa con ellas correctamente.

6.5.1 El paquete `mockito`

`mockito` es un popular paquete de mocking para Dart y Flutter que te permite crear mocks de clases y definir su comportamiento esperado. Es muy útil para probar clases que dependen de otras clases.

Instalación:

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  mockito: ^5.4.4 # Usa la última versión estable
  build_runner: ^2.4.8 # Necesario para generar los mocks
```

Luego, ejecuta `flutter pub get`.

Uso:

1. **Anota la clase a mockear**: Añade `@GenerateMocks([ClaseAReemplazar])` encima de la función `main()` de tu archivo de prueba.

2. **Ejecuta el generador de código**: Ejecuta `flutter pub run build_runner build` (o `watch`) para generar el archivo mock.

Ejemplo de mocking con `mockito`:

Supongamos que tenemos un servicio de datos que obtiene usuarios de una API y queremos probar un `UserManager` que depende de este servicio.

```dart
// lib/services/user_service.dart
class User {
  final String id;
  final String name;

  User({required this.id, required this.name});
}

class UserService {
  Future<User> fetchUser(String userId) async {
    // Simula una llamada a la API
    await Future.delayed(const Duration(seconds: 1));
    if (userId == '123') {
      return User(id: userId, name: 'Alice');
    } else {
      throw Exception('Usuario no encontrado');
    }
  }
}

// lib/user_manager.dart
class UserManager {
  final UserService _userService;

  UserManager(this._userService);

  Future<String> getUserName(String userId) async {
    try {
      final user = await _userService.fetchUser(userId);
      return user.name;
    } catch (e) {
      return 'Error: ${e.toString()}';
    }
  }
}

// test/user_manager_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';
import 'package:tu_app/services/
user_service.dart'; // Asegúrate de que la ruta sea correcta
```

```dart
import 'package:tu_app/user_manager.dart'; // Asegúrate de que la ruta sea correcta

import 'user_manager_test.mocks.dart'; // Este archivo será generado

@GenerateMocks([UserService])
void main() {
  group('UserManager', () {
    late MockUserService mockUserService; // Instancia del mock
    late UserManager userManager;

    setUp(() {
      mockUserService = MockUserService();
      userManager = UserManager(mockUserService);
    });

    test('getUserName debe retornar el nombre del usuario si la llamada es exitosa', () async {
      // Define el comportamiento del mock cuando se llama a fetchUser
      when(mockUserService.fetchUser('123'))
          .thenAnswer((_) async => User(id: '123', name: 'Bob'));

      final name = await userManager.getUserName('123');
      expect(name, 'Bob');

      // Verifica que fetchUser fue llamado exactamente una vez con '123'
      verify(mockUserService.fetchUser('123')).called(1);
    });

    test('getUserName debe retornar un mensaje de error si la llamada falla', () async {
      // Define el comportamiento del mock para lanzar una excepción
      when(mockUserService.fetchUser('456'))
          .thenThrow(Exception('Error de red'));

      final name = await userManager.getUserName('456');
      expect(name, contains('Error de red'));

      // Verifica que fetchUser fue llamado exactamente una vez con '456'
      verify(mockUserService.fetchUser('456')).called(1);
    });
  });
}
```

Después de guardar `user_manager_test.dart`, ejecuta `flutter pub run build_runner build` para generar `user_manager_test.mocks.dart`.

Explicación:

- **`@GenerateMocks([UserService])`**: Esta anotación le dice a `build_runner` que genere una clase mock para `UserService`.
- **`import 'user_manager_test.mocks.dart';`**: Importa el archivo generado que contiene la clase `MockUserService`.
- **`MockUserService mockUserService;`**: Declara una instancia de la clase mock.
- **`when(mockUserService.fetchUser('123')).thenAnswer((_) async => User(id: '123', name: 'Bob'));`**: Esta es la parte clave del mocking. Le dices a `mockito` que cuando se llame a `mockUserService.fetchUser` con el argumento `'123'`, debe devolver un `Future` que se resuelve con un `User` con nombre 'Bob'.
- **`verify(mockUserService.fetchUser('123')).called(1);`**: Después de ejecutar el código bajo prueba, puedes verificar que los métodos del mock fueron llamados como esperabas.

6.5.2 Simulación de dependencias en pruebas de widgets

En las pruebas de widgets, a menudo necesitas simular dependencias como `ChangeNotifier`s (para `provider`), `Bloc`s (para `flutter_bloc`), o servicios de red. Esto se hace envolviendo el widget bajo prueba con los proveedores o mocks necesarios.

Ejemplo con `provider` y `mockito`:

Si tu widget depende de un `ChangeNotifier` proporcionado por `provider`, puedes mockearlo y proporcionarlo en tu prueba de widget.

```dart
// lib/contador_modelo.dart (del Capítulo 3)
import 'package:flutter/foundation.dart';

class ContadorModelo extends ChangeNotifier {
  int _conteo = 0;
  int get conteo => _conteo;
  void incrementar() {
    _conteo++;
    notifyListeners();
  }
}

// lib/widgets/contador_display.dart
```

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:tu_app/contador_modelo.dart';

class ContadorDisplay extends StatelessWidget {
  const ContadorDisplay({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final conteo = context.watch<ContadorModelo>().conteo;
    return Text(
      'Contador: $conteo',
      style: Theme.of(context).textTheme.headlineMedium,
    );
  }
}

// test/widgets/contador_display_test.dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';
import 'package:provider/provider.dart';
import 'package:tu_app/contador_modelo.dart';
import 'package:tu_app/widgets/contador_display.dart';

import 'contador_display_test.mocks.dart';

@GenerateMocks([ContadorModelo])
void main() {
  group('ContadorDisplay', () {
    late MockContadorModelo mockContadorModelo;

    setUp(() {
      mockContadorModelo = MockContadorModelo();
    });

    testWidgets('muestra el valor correcto del contador',
(WidgetTester tester) async {
      // Define el comportamiento del mock
      when(mockContadorModelo.conteo).thenReturn(5);

      // Envuelve el widget bajo prueba con el
ChangeNotifierProvider mockeado
      await tester.pumpWidget(
        MaterialApp(
          home: ChangeNotifierProvider<ContadorModelo>.value(
            value: mockContadorModelo,
            child: const ContadorDisplay(),
          ),
        ),
      );
```

```
      // Verifica que el texto del contador es 'Contador: 5'
      expect(find.text('Contador: 5'), findsOneWidget);
    });

    testWidgets('se actualiza cuando el contador cambia',
(WidgetTester tester) async {
      // Define el comportamiento inicial del mock
      when(mockContadorModelo.conteo).thenReturn(0);

      await tester.pumpWidget(
        MaterialApp(
          home: ChangeNotifierProvider<ContadorModelo>.value(
            value: mockContadorModelo,
            child: const ContadorDisplay(),
          ),
        ),
      );

      expect(find.text('Contador: 0'), findsOneWidget);

      // Simula un cambio en el mock y notifica a los oyentes
      when(mockContadorModelo.conteo).thenReturn(1);
      mockContadorModelo.notifyListeners(); // Esto es crucial
para que el Consumer se reconstruya

      await tester.pump(); // Dispara un frame para que los
cambios se reflejen

      expect(find.text('Contador: 1'), findsOneWidget);
    });
  });
}
```

Después de guardar `contador_display_test.dart`, ejecuta `flutter pub run build_runner build` para generar `contador_display_test.mocks.dart`.

El mocking y la simulación son herramientas poderosas que te permiten escribir pruebas más rápidas, fiables y aisladas, lo que es fundamental para mantener la calidad del código en aplicaciones Flutter.

Capítulo 7: Despliegue de Aplicaciones

Una vez que tu aplicación Flutter está desarrollada y probada, el siguiente paso crucial es distribuirla a tus usuarios. El proceso de despliegue varía ligeramente según la plataforma de destino (Android, iOS, Web, Desktop), pero Flutter simplifica gran parte de este trabajo al permitirte generar paquetes para cada una desde una única base de

código. Este capítulo te guiará a través de los pasos esenciales para preparar y desplegar tu aplicación Flutter en las principales tiendas de aplicaciones y plataformas.

7.1 Preparación para el despliegue (iconos, splash screen)

Antes de generar los paquetes de lanzamiento, es importante realizar algunas configuraciones y añadir activos visuales que mejoran la experiencia del usuario y la profesionalidad de tu aplicación.

7.1.1 Iconos de la aplicación

El icono de la aplicación es lo primero que los usuarios ven en sus pantallas de inicio y en las tiendas de aplicaciones. Es fundamental que sea atractivo y represente tu marca. Flutter no genera automáticamente los iconos para todas las plataformas, por lo que necesitarás herramientas o paquetes para simplificar este proceso.

Uso del paquete `flutter_launcher_icons`:

Este paquete automatiza la generación de iconos de aplicación para Android e iOS a partir de una única imagen fuente.

1. **Añadir la dependencia**: En tu `pubspec.yaml`: ```yaml dev_dependencies: flutter_launcher_icons: ^0.13.1 # Usa la última versión estable

   flutter_launcher_icons: android: "launcher_icon" ios: true image_path: "assets/icon/app_icon.png" # Ruta a tu imagen fuente (PNG, JPG) min_sdk_android: 21 # Versión mínima de Android soportada # adaptive_icon_background: "#FFFFFF" # Opcional: color de fondo para iconos adaptativos de Android # adaptive_icon_foreground: "assets/icon/app_icon_foreground.png" # Opcional: imagen de primer plano para iconos adaptativos ```

2. **Preparar la imagen fuente**: Coloca tu imagen de icono (preferiblemente un PNG cuadrado de alta resolución, por ejemplo, 1024x1024 píxeles) en la ruta especificada (ej. `assets/icon/app_icon.png`).

3. **Generar los iconos**: Ejecuta el siguiente comando en tu terminal: `bash flutter pub run flutter_launcher_icons` Esto generará los iconos en los directorios nativos (`android/app/src/main/res/` y `ios/Runner/Assets.xcassets/`).

7.1.2 Pantalla de bienvenida (Splash Screen)

La pantalla de bienvenida (splash screen o launch screen) es la primera pantalla que ven los usuarios cuando abren tu aplicación. Se muestra mientras la aplicación se carga. Una

buena pantalla de bienvenida mejora la percepción de la aplicación y reduce la sensación de espera.

Uso del paquete `flutter_native_splash`:

Este paquete te permite generar una pantalla de bienvenida nativa para Android, iOS y Web.

1. **Añadir la dependencia**: En tu `pubspec.yaml`: ```yaml dev_dependencies: flutter_native_splash: ^2.3.10 # Usa la última versión estable

   flutter_native_splash: color: "#FFFFFF" # Color de fondo de la splash screen image: "assets/splash/splash_image.png" # Ruta a tu imagen para la splash screen # brand_image: "assets/splash/brand_logo.png" # Opcional: imagen de marca en la parte inferior # android: true # ios: true # web: false # android_gravity: center # ios_content_mode: center # web_image_mode: center ```

2. **Preparar la imagen fuente**: Coloca tu imagen para la splash screen (por ejemplo, un PNG) en la ruta especificada (ej. `assets/splash/splash_image.png`).

3. **Generar la splash screen**: Ejecuta el siguiente comando en tu terminal: `bash flutter pub run flutter_native_splash:create` Esto modificará los archivos nativos para incluir tu splash screen.

   Para eliminar la splash screen generada: `bash flutter pub run flutter_native_splash:remove`

7.1.3 Nombre de la aplicación y versión

- **Nombre de la aplicación**: Puedes cambiar el nombre visible de tu aplicación en `pubspec.yaml` bajo la sección `name` y también en los archivos nativos (`android/app/src/main/AndroidManifest.xml` y `ios/Runner/Info.plist`).
- **Versión**: La versión de tu aplicación se define en `pubspec.yaml` en el campo `version` (ej. `1.0.0+1`). El primer número (`1.0.0`) es la versión visible para el usuario, y el número después del `+` (`1`) es el código de compilación, que debe ser único y creciente para cada lanzamiento.

7.1.4 Firmado de la aplicación

Para desplegar tu aplicación en las tiendas, necesitarás firmarla digitalmente. Esto asegura la integridad de tu aplicación y verifica su origen.

- **Android**: Necesitarás generar una clave de firma (`keystore`) y configurar tu proyecto Android para usarla. Esto se hace en el archivo `android/app/build.gradle` .
- **iOS**: Necesitarás un certificado de firma de Apple y un perfil de aprovisionamiento, gestionados a través de Xcode y tu cuenta de desarrollador de Apple.

Estos pasos son críticos para la preparación de tu aplicación antes de su distribución.

7.2 Despliegue en Android (Google Play Store)

El despliegue de una aplicación Flutter en Google Play Store implica varios pasos, desde la preparación del paquete de lanzamiento hasta la configuración en la consola de Google Play.

7.2.1 Generar un paquete de lanzamiento (APK/App Bundle)

Flutter puede generar un APK (Android Package Kit) o un AAB (Android App Bundle). Se recomienda encarecidamente usar App Bundles, ya que Google Play los utiliza para generar APKs optimizados para la configuración de cada dispositivo, lo que resulta en descargas más pequeñas para los usuarios.

1. **Generar una clave de firma (si no tienes una)**: `bash keytool -genkey -v -keystore ~/upload-keystore.jks -keyalg RSA -keysize 2048 -validity 10000 -alias upload` Esto creará un archivo `upload-keystore.jks` en tu directorio de inicio. Recuerda la contraseña que uses y el alias (`upload`).

2. **Configurar la firma en tu proyecto Flutter**: Crea un archivo `android/key.properties` con el siguiente contenido (reemplaza con tus datos): `properties storePassword=TU_CONTRASEÑA_DEL_KEYSTORE keyPassword=TU_CONTRASEÑA_DE_LA_CLAVE keyAlias=upload storeFile=/home/tu_usuario/upload-keystore.jks # Ruta completa a tu keystore`

   Luego, edita `android/app/build.gradle` para usar esta configuración de firma. Busca la sección `android { ... }` y añade:

```gradle android { // ... signingConfigs { release { storeFile file(System.getenv("FLUTTER_KEYSTORE_PATH") ?: "key.properties") storePassword System.getenv("FLUTTER_KEYSTORE_PASSWORD") ?: "" keyAlias System.getenv("FLUTTER_KEY_ALIAS") ?: "" keyPassword System.getenv("FLUTTER_KEY_PASSWORD") ?: "" } }

```
buildTypes {
    release {
        signingConfig signingConfigs.release
        // ...
    }
}
```

} `` **Nota**: Es una buena práctica usar variables de entorno para las contraseñas en entornos de CI/CD. Para desarrollo local, key.properties` es suficiente.

3. **Generar el App Bundle**: En la raíz de tu proyecto Flutter, ejecuta: `bash flutter build appbundle` El archivo `.aab` se generará en `build/app/outputs/bundle/release/app-release.aab`.

#### 7.2.2 Subir a Google Play Console

1. **Crear una cuenta de desarrollador**: Si aún no tienes una, regístrate en Google Play Console (play.google.com/console/developers) y paga la tarifa de registro única.
2. **Crear una nueva aplicación**: En Google Play Console, haz clic en "Crear aplicación" y sigue los pasos para proporcionar la información básica de tu aplicación.
3. **Configurar el lanzamiento**: En el menú de la izquierda, ve a "Versión" > "Producción" (o "Pruebas internas", "Pruebas cerradas", etc., para pruebas) y haz clic en "Crear nueva versión".
4. **Subir el App Bundle**: Sube el archivo `app-release.aab` que generaste.
5. **Completar la información de la ficha de Play Store**: Proporciona detalles como el título, la descripción, las capturas de pantalla, el icono de alta resolución, la imagen de la función, el tipo de aplicación, la categoría, la política de privacidad, etc.
6. **Clasificación de contenido**: Completa el cuestionario de clasificación de contenido para obtener una clasificación adecuada para tu aplicación.
```

7. **Precios y distribución**: Define si tu aplicación es gratuita o de pago y a qué países quieres distribuirla.

8. **Revisar y lanzar**: Revisa toda la información y, una vez que estés satisfecho, envía la aplicación para revisión. Google revisará tu aplicación para asegurarse de que cumple con sus políticas. Una vez aprobada, se publicará en Google Play Store.

7.3 Despliegue en iOS (Apple App Store)

El despliegue de una aplicación Flutter en la Apple App Store es un proceso más riguroso que requiere una cuenta de desarrollador de Apple y el uso de Xcode.

7.3.1 Requisitos previos

- **Mac con Xcode**: Necesitas una computadora Mac con Xcode instalado para construir y firmar tu aplicación iOS.
- **Cuenta de desarrollador de Apple**: Debes tener una suscripción activa al Apple Developer Program (developer.apple.com/programs/), que tiene una tarifa anual.

7.3.2 Configuración del proyecto iOS en Xcode

1. **Abrir el proyecto iOS en Xcode**: Navega a la carpeta `ios/Runner.xcworkspace` en tu proyecto Flutter y ábrela con Xcode.

2. **Configurar la firma automática**: En Xcode, selecciona tu proyecto en el navegador de proyectos, luego ve a la pestaña "Signing & Capabilities". Asegúrate de que "Automatically manage signing" esté marcado y selecciona tu "Team" (tu cuenta de desarrollador de Apple).

3. **Bundle Identifier**: Asegúrate de que el "Bundle Identifier" (en la pestaña "General" de tu objetivo Runner) sea único y coincida con el ID de tu aplicación en App Store Connect (ej. `com.tuempresa.tuapp`).

4. **Versión y Build**: En la pestaña "General", verifica que los campos "Version" y "Build" coincidan con la versión y el código de compilación definidos en tu `pubspec.yaml`.

7.3.3 Generar un archivo de archivo (IPA)

1. **Seleccionar un dispositivo genérico**: En Xcode, selecciona "Any iOS Device (arm64)" como el dispositivo de destino.

2. **Archivar la aplicación**: Ve a `Product` > `Archive` en la barra de menú de Xcode. Esto compilará tu aplicación y creará un archivo (`.xcarchive`).

3. **Distribuir la aplicación**: Una vez que el archivado se complete, se abrirá la ventana "Organizer". Selecciona tu archivo y haz clic en "Distribute App".

4. **Elegir el método de distribución**: Selecciona "App Store Connect" y sigue los pasos. Xcode te guiará a través del proceso de firma y subida a App Store Connect.

7.3.4 Subir a App Store Connect

1. **Crear una nueva aplicación**: Inicia sesión en App Store Connect (appstoreconnect.apple.com) y crea una nueva aplicación. Proporciona el nombre, el Bundle ID, la SKU y la versión.

2. **Configurar la información de la aplicación**: Completa todos los metadatos requeridos, como el nombre, la categoría, la política de privacidad, la clasificación por edad, las capturas de pantalla para diferentes tamaños de dispositivo, el icono de la aplicación (1024x1024 píxeles), y la descripción.

3. **Seleccionar la compilación**: Una vez que Xcode haya subido tu archivo, aparecerá en la sección "Build" de tu versión en App Store Connect. Selecciónalo.

4. **Revisar y enviar**: Revisa toda la información cuidadosamente. Una vez que todo esté configurado, haz clic en "Submit for Review". Apple revisará tu aplicación para asegurarse de que cumple con sus estrictas directrices. Este proceso puede tardar varios días.

Una vez aprobada, tu aplicación estará disponible en la Apple App Store.

7.4 Despliegue en Web y Desktop

Flutter también permite desplegar aplicaciones en la web y como aplicaciones de escritorio nativas. Aunque el proceso es generalmente más sencillo que el despliegue en tiendas móviles, hay consideraciones específicas para cada plataforma.

7.4.1 Despliegue Web

Las aplicaciones Flutter Web se compilan a HTML, CSS y JavaScript, y pueden ser alojadas en cualquier servidor web. Flutter soporta dos renderizadores web: HTML (para compatibilidad) y CanvasKit (para rendimiento).

1. **Habilitar el soporte web**: Si no lo has hecho, habilita el soporte web para tu proyecto:
   `bash flutter config --enable-web flutter create . # Para añadir los archivos web al proyecto existente`

2. **Compilar para la web**: Ejecuta el siguiente comando para generar los archivos de compilación web: `bash flutter build web` Los archivos generados se encontrarán en el directorio `build/web/`.

3. **Alojar los archivos**: Sube el contenido del directorio `build/web/` a tu servidor web preferido (por ejemplo, Firebase Hosting, Netlify, GitHub Pages, Nginx, Apache). Asegúrate de que el servidor esté configurado para servir `index.html` como el archivo predeterminado.

   Ejemplo con Firebase Hosting: * Instala Firebase CLI: `npm install -g firebase-tools` * Inicia sesión: `firebase login` * Inicializa Firebase en tu proyecto: `firebase init` (selecciona Hosting y apunta al directorio `build/web`) * Despliega: `firebase deploy --only hosting`

7.4.2 Despliegue en Desktop (Windows, macOS, Linux)

Flutter puede compilar aplicaciones nativas para Windows, macOS y Linux. El proceso de compilación genera ejecutables y paquetes específicos para cada sistema operativo.

1. **Habilitar el soporte de escritorio**: Si no lo has hecho, habilita el soporte para la plataforma deseada: `bash flutter config --enable-windows-desktop flutter config --enable-macos-desktop flutter config --enable-linux-desktop flutter create . # Para añadir los archivos de escritorio al proyecto existente`

2. **Compilar para la plataforma de escritorio**: Ejecuta el siguiente comando para generar el ejecutable para la plataforma actual: `bash flutter build windows flutter build macos flutter build linux` Los ejecutables y los archivos de soporte se encontrarán en `build/windows/runner/Release/`, `build/macos/Build/Products/Release/`, o `build/linux/x64/release/bundle/` respectivamente.

3. **Empaquetado y distribución**: El proceso de empaquetado para la distribución final varía según el sistema operativo. Puedes crear instaladores (MSI para Windows, DMG para macOS, .deb/.rpm para Linux) utilizando herramientas específicas de cada plataforma o de terceros. Flutter no proporciona una herramienta integrada para crear instaladores complejos, pero los archivos generados son aplicaciones nativas que pueden ser distribuidas.

   - **Windows**: Los archivos generados son un ejecutable (`.exe`) y sus dependencias. Puedes comprimirlos en un archivo ZIP o usar herramientas como Inno Setup o NSIS para crear un instalador.

    - **macOS**: Se genera un archivo `.app` que se puede empaquetar en un archivo `.dmg` para su distribución.
    - **Linux**: Se genera un paquete de aplicación que puede ser distribuido como un archivo `.deb` (Debian/Ubuntu), `.rpm` (Fedora/RHEL) o un AppImage.

El despliegue en web y escritorio abre tu aplicación Flutter a una audiencia más amplia, aprovechando la capacidad multiplataforma de Flutter al máximo.

Capítulo 8: Optimización y Rendimiento

El rendimiento es un factor crítico para el éxito de cualquier aplicación. Una aplicación lenta, con animaciones entrecortadas o que consume demasiados recursos, puede frustrar a los usuarios y llevar a una mala experiencia. Flutter, al compilar a código nativo y tener un motor de renderizado propio, ofrece un rendimiento excepcional, pero aún así es fundamental entender cómo optimizar tu código y tu UI para asegurar la máxima fluidez y eficiencia. Este capítulo explorará herramientas y técnicas para analizar, identificar y resolver problemas de rendimiento en tus aplicaciones Flutter.

8.1 Análisis de rendimiento (DevTools)

Flutter DevTools es un conjunto de herramientas de depuración y rendimiento que te ayudan a inspeccionar, perfilar y optimizar tus aplicaciones Flutter. Es una herramienta indispensable para cualquier desarrollador de Flutter.

Acceso a DevTools:

Puedes abrir DevTools de varias maneras:

- **Desde tu IDE**: Tanto Android Studio como Visual Studio Code tienen integraciones que te permiten lanzar DevTools directamente desde la barra de herramientas de depuración.
- **Desde la línea de comandos**: Ejecuta `flutter pub global activate devtools` (si no lo tienes activado) y luego `flutter pub global run devtools` en tu terminal. Esto abrirá DevTools en tu navegador web.

Secciones clave de DevTools para rendimiento:

- **Inspector de Widgets**: Te permite visualizar el árbol de widgets de tu aplicación, inspeccionar las propiedades de los widgets y entender cómo se componen. Es útil para identificar widgets innecesarios o anidados profundamente que podrían afectar el rendimiento.

- **Performance (Rendimiento)**: Esta sección es crucial para identificar problemas de rendimiento. Muestra:
    - **Gráficos de FPS (Frames Per Second)**: Muestra la tasa de fotogramas de la UI y del GPU. Un valor constante de 60 FPS (o 120 FPS en dispositivos compatibles) indica una UI fluida. Caídas por debajo de este valor sugieren problemas de rendimiento.
    - **Timeline (Línea de tiempo)**: Muestra los eventos de renderizado de la UI y del GPU, permitiéndote ver qué operaciones están tomando más tiempo. Puedes grabar sesiones para analizar el rendimiento de flujos específicos de tu aplicación.
    - **CPU Profiler (Perfilador de CPU)**: Muestra el uso de la CPU por tu aplicación, ayudándote a identificar cuellos de botella en tu código Dart.
- **Memory (Memoria)**: Te permite inspeccionar el uso de memoria de tu aplicación, identificar fugas de memoria y objetos que no se están liberando correctamente.
- **Network (Red)**: Muestra las solicitudes de red realizadas por tu aplicación, incluyendo el tiempo que tardan y el tamaño de la respuesta. Útil para optimizar el consumo de APIs.

Cómo usar DevTools para identificar problemas:

1. **Ejecuta tu aplicación en modo de depuración o perfil**: Para obtener datos de rendimiento precisos, ejecuta tu aplicación con `flutter run --profile` o `flutter run --release` (aunque el modo de perfil es mejor para depuración de rendimiento).
2. **Observa los gráficos de FPS**: Si los FPS caen por debajo de 60 (o 120), hay un problema de rendimiento.
3. **Usa el Timeline**: Graba una sesión mientras interactúas con la parte de la aplicación que tiene problemas de rendimiento. Busca "jank" (fotogramas perdidos) en el timeline y examina los eventos que ocurren durante esos picos. Esto te ayudará a identificar qué operaciones están causando la lentitud.
4. **Usa el CPU Profiler**: Si el timeline indica que el problema está en la CPU, usa el profiler para ver qué funciones de tu código están consumiendo más tiempo.
5. **Usa el Inspector de Widgets**: Busca árboles de widgets muy profundos o widgets que se reconstruyen innecesariamente. Las reconstrucciones innecesarias son una causa común de problemas de rendimiento.

DevTools es tu mejor amigo para entender el comportamiento de tu aplicación y encontrar oportunidades de optimización.

8.2 Optimización de la UI (reconstrucciones, árboles de widgets)

La optimización de la interfaz de usuario en Flutter se centra principalmente en minimizar las reconstrucciones innecesarias de widgets y en construir árboles de widgets eficientes. Cada vez que un widget se reconstruye, Flutter debe volver a calcular su layout y redibujarlo, lo que puede consumir recursos si se hace en exceso.

8.2.1 Minimizar reconstrucciones innecesarias

- **Usar `const` siempre que sea posible**: Si un widget y todos sus descendientes son inmutables y no dependen de ningún estado cambiante, decláralos como `const`. Flutter puede optimizar el renderizado de widgets `const` al no reconstruirlos si sus propiedades no cambian. ```dart // Mal Widget build(BuildContext context) { return Text("Hola Mundo"); }

  // Bien Widget build(BuildContext context) { return const Text("Hola Mundo"); } ```

- **Limitar el alcance de `setState`**: `setState` reconstruye todo el `StatefulWidget` y sus descendientes. Si solo una pequeña parte de tu UI necesita actualizarse, intenta encapsular esa parte en un `StatefulWidget` separado y llama a `setState` solo en ese widget.

- **Usar `Consumer` o `Selector` con `provider` (o `BlocBuilder` con BLoC)**: Cuando uses soluciones de gestión de estado como `provider` o BLoC, utiliza los widgets `Consumer` o `BlocBuilder` para escuchar solo los cambios de estado relevantes. Esto asegura que solo la parte mínima del árbol de widgets se reconstruya cuando el estado cambia. ```dart // Mal (reconstruye todo el widget cuando el contador cambia) class MyPage extends StatelessWidget { @override Widget build(BuildContext context) { final contador = context.watch().conteo; return Column(children: [Text("Contador: $contador"), // Otros widgets que no dependen del contador SomeOtherWidget(),],); } }

  // Bien (solo reconstruye el Text cuando el contador cambia) class MyPage extends StatelessWidget { @override Widget build(BuildContext context) { return Column(children: [Consumer(builder: (context, contadorModelo, child) { return Text("Contador: ${contadorModelo.conteo}"); },), // Otros widgets que no dependen del contador const SomeOtherWidget(), // Si es posible, hazlo const],); } } ```

- **Evitar crear widgets en el método `build` de forma innecesaria**: Si un widget no cambia, no lo crees dentro del método `build` de un `StatefulWidget` si puede

ser una constante o una variable de instancia. Esto evita que se cree una nueva instancia del widget en cada reconstrucción.

8.2.2 Optimización del árbol de widgets

- **Aplanar el árbol de widgets**: Un árbol de widgets muy profundo puede afectar el rendimiento del layout. Intenta usar widgets que combinen funcionalidades (por ejemplo, `ListTile` en lugar de `Row` con `Column` y `Text`s separados) o refactoriza widgets complejos en componentes más pequeños y reutilizables.
- **Usar `Sliver`s para listas grandes**: Para listas con muchos elementos, `ListView` es eficiente porque solo renderiza los elementos visibles. Sin embargo, para efectos de desplazamiento personalizados o listas muy grandes, considera usar `CustomScrollView` con `Sliver`s (como `SliverList` o `SliverGrid`). Los `Sliver`s ofrecen un control más granular sobre el comportamiento de desplazamiento y el renderizado.
- **`RepaintBoundary`**: Si tienes un widget complejo que se reconstruye con frecuencia pero su contenido visual no cambia, puedes envolverlo en un `RepaintBoundary`. Esto le dice a Flutter que puede almacenar en caché la capa de renderizado de ese widget, evitando que se redibuje innecesariamente. Úsalo con precaución, ya que puede tener un costo de memoria.
- **`Opacity` vs `AnimatedOpacity`**: Si solo necesitas cambiar la opacidad de un widget, usa `AnimatedOpacity` en lugar de `Opacity` dentro de un `setState`. `Opacity` fuerza una reconstrucción de su hijo, mientras que `AnimatedOpacity` utiliza una optimización de capa para animar la opacidad sin reconstruir el hijo.

Al aplicar estas técnicas, puedes reducir la carga de trabajo de renderizado de Flutter y asegurar que tu aplicación se mantenga fluida y receptiva, incluso en dispositivos menos potentes.

8.3 Optimización de la red y el almacenamiento

La interacción con la red y el almacenamiento local son operaciones que pueden impactar significativamente el rendimiento y la experiencia del usuario. Optimizar estas áreas es crucial para una aplicación eficiente.

8.3.1 Optimización de la red

- **Reducir el tamaño de los datos**: Envía y recibe solo los datos necesarios. Utiliza compresión (gzip) si tu API lo soporta. Para imágenes, usa formatos eficientes

(WebP) y comprímelas antes de subirlas o servir versiones optimizadas para diferentes tamaños de pantalla.

- **Caché de red**: Implementa una estrategia de caché para los datos de red. Esto reduce el número de solicitudes a la API y permite que la aplicación funcione sin conexión o cargue datos más rápido. Paquetes como `dio_http_cache` o la gestión manual de caché pueden ser útiles.
- **Paginación**: Para listas grandes de datos, implementa paginación en tus llamadas a la API. Carga solo un subconjunto de datos a la vez (por ejemplo, 20 elementos) y carga más a medida que el usuario se desplaza. Esto reduce el uso de memoria y el tiempo de carga inicial.
- **Manejo de errores y reintentos**: Implementa un manejo robusto de errores de red y estrategias de reintento con retroceso exponencial para solicitudes fallidas. Esto mejora la resiliencia de tu aplicación.
- **Pre-fetching de datos**: Si sabes que el usuario probablemente necesitará ciertos datos en el futuro cercano, puedes precargarlos en segundo plano para que estén disponibles cuando se necesiten.
- **WebSockets o Streams para datos en tiempo real**: Para datos que cambian con mucha frecuencia (por ejemplo, chats, actualizaciones de precios), considera usar WebSockets o soluciones basadas en streams (como Firestore) en lugar de polling HTTP repetitivo, que es menos eficiente.

8.3.2 Optimización del almacenamiento

- **Elegir la solución de almacenamiento adecuada**: Como se vio en el Capítulo 5, selecciona la solución de persistencia local que mejor se adapte a tus necesidades (`shared_preferences` para datos simples, `Hive` para datos NoSQL, `sqflite` para datos relacionales). Usar la herramienta incorrecta puede llevar a ineficiencias.
- **Almacenar solo lo necesario**: Evita almacenar datos redundantes o innecesarios en el dispositivo. Limpia los datos antiguos o no utilizados regularmente.
- **Indexación de bases de datos**: Si usas `sqflite` , asegúrate de que las columnas que utilizas en tus cláusulas `WHERE` o `ORDER BY` estén indexadas para acelerar las consultas.
- **Serialización/Deserialización eficiente**: Para datos complejos que se almacenan localmente, utiliza `json_serializable` o `Hive` con sus adaptadores generados para una serialización y deserialización rápida y eficiente.

- **Manejo de archivos grandes**: Si tu aplicación maneja archivos grandes (imágenes, videos), considera almacenarlos en el almacenamiento externo del dispositivo y gestiona su ciclo de vida (eliminación cuando ya no se necesitan).
- **Compresión de datos**: Para datos almacenados localmente, considera la compresión si el tamaño es una preocupación y el acceso no es extremadamente frecuente.

Al optimizar la forma en que tu aplicación interactúa con la red y gestiona el almacenamiento, puedes mejorar significativamente la velocidad de carga, la capacidad de respuesta y la eficiencia general de tu aplicación, lo que se traduce en una mejor experiencia para el usuario.

8.4 Buenas prácticas para un código eficiente

Además de las optimizaciones específicas de UI, red y almacenamiento, adoptar buenas prácticas de codificación en Dart y Flutter es fundamental para escribir aplicaciones eficientes y de alto rendimiento desde el principio.

8.4.1 Evitar operaciones costosas en el método `build`

El método `build` de un widget puede ser llamado muchas veces, especialmente durante animaciones o cuando el estado cambia. Por lo tanto, es crucial que el código dentro de `build` sea lo más ligero y rápido posible.

- **No realizar cálculos complejos**: Evita operaciones que consuman mucho tiempo, como cálculos matemáticos intensivos, llamadas a la red, o consultas a bases de datos dentro del método `build`. Realiza estas operaciones en `initState`, `didUpdateWidget`, o en métodos asíncronos y actualiza el estado con `setState` (o tu solución de gestión de estado) cuando los resultados estén listos.
- **No crear objetos innecesariamente**: Evita crear nuevas instancias de objetos (como `List`, `Map`, `DateTime`) en cada llamada a `build` si no es necesario. Si un objeto es constante, decláralo como `const`. Si es una variable de instancia, inicialízala una vez.
- **Usar `final` y `const`**: Declara variables como `final` o `const` siempre que sea posible. Esto no solo mejora la legibilidad y la seguridad del código, sino que también permite a Dart y Flutter realizar optimizaciones de rendimiento.

8.4.2 Uso eficiente de colecciones

- **Elegir la colección adecuada**: Utiliza la colección de Dart más apropiada para tu caso de uso (`List`, `Set`, `Map`). Por ejemplo, `Set` es eficiente para verificar la existencia de elementos, y `Map` para búsquedas por clave.
- **Evitar operaciones costosas en bucles**: Si tienes bucles que iteran sobre colecciones grandes, asegúrate de que las operaciones dentro del bucle sean eficientes. Evita llamadas a métodos que recorran la colección repetidamente.
- **`Iterable` vs `List`**: Cuando trabajes con colecciones, utiliza `Iterable` cuando sea posible para operaciones que no necesiten una lista concreta. `Iterable`s son "lazy" (perezosos), lo que significa que las operaciones se realizan solo cuando se necesitan, lo que puede ahorrar memoria y tiempo de procesamiento.

8.4.3 Asincronía y concurrencia

- **Usar `async` / `await` para operaciones no bloqueantes**: Las operaciones de E/S (red, disco) y los cálculos intensivos deben ejecutarse de forma asíncrona para evitar bloquear el hilo principal de la UI y causar "jank". Utiliza `async` y `await` para manejar estas operaciones de forma limpia.
- **`Isolates` para cálculos intensivos**: Para cálculos que consumen mucha CPU y que no pueden ser asíncronos (es decir, que bloquearían el hilo principal), utiliza `Isolates`. Los `Isolates` son como hilos separados que no comparten memoria, lo que evita que bloqueen la UI. El paquete `flutter_isolate` puede ser útil para esto.

8.4.4 Optimización de imágenes

- **Cargar imágenes en el tamaño correcto**: No cargues imágenes de alta resolución si solo se van a mostrar en un tamaño pequeño. Utiliza `Image.network` con `width` y `height` explícitos, o pídele a tu servidor que sirva imágenes en el tamaño adecuado.
- **Caché de imágenes**: Flutter tiene un caché de imágenes incorporado. Asegúrate de que tus imágenes se estén almacenando en caché correctamente para evitar descargas repetidas.
- **Formatos de imagen eficientes**: Utiliza formatos de imagen modernos y eficientes como WebP cuando sea posible, ya que ofrecen una mejor compresión sin pérdida de calidad significativa.

8.4.5 Perfilado y depuración continuos

- **Pruebas de rendimiento regulares**: Integra pruebas de rendimiento en tu proceso de desarrollo. Utiliza DevTools regularmente para perfilar tu aplicación a medida que añades nuevas características.
- **Monitoreo en producción**: Considera integrar herramientas de monitoreo de rendimiento (como Firebase Performance Monitoring) en tu aplicación de producción para identificar cuellos de botella en el mundo real.

Al seguir estas buenas prácticas, puedes construir aplicaciones Flutter que no solo sean funcionales y atractivas, sino también rápidas, fluidas y eficientes en el uso de recursos.

Capítulo 9: Integración con Funcionalidades Nativas

Aunque Flutter es una plataforma multiplataforma que te permite escribir la mayor parte de tu código en Dart, hay ocasiones en las que necesitas acceder a funcionalidades específicas de la plataforma (Android o iOS) que no están disponibles directamente a través de los paquetes de Flutter. Esto puede incluir el uso de APIs de hardware (cámara, GPS, Bluetooth), frameworks específicos del sistema operativo, o bibliotecas nativas existentes. Flutter proporciona un mecanismo robusto para comunicarse con el código nativo: los Platform Channels.

9.1 Platform Channels

Los Platform Channels son el puente que permite la comunicación bidireccional entre el código Dart de tu aplicación Flutter y el código nativo de la plataforma (Kotlin/Java en Android, Swift/Objective-C en iOS). Funcionan enviando mensajes a través de un "canal" específico.

Componentes clave de los Platform Channels:

- **`MethodChannel`**: Se utiliza para invocar métodos en la plataforma desde Dart y recibir resultados, o para invocar métodos en Dart desde la plataforma.
- **`EventChannel`**: Se utiliza para enviar flujos de eventos continuos desde la plataforma a Dart (por ejemplo, actualizaciones de sensores o de ubicación).
- **`BasicMessageChannel`**: Para el intercambio de mensajes asíncronos y bidireccionales de tipo arbitrario.

El `MethodChannel` es el más comúnmente utilizado para la mayoría de las interacciones nativas.

9.1.1 Cómo funciona `MethodChannel`

1. **Dart (Cliente)**: El código Dart invoca un método en el `MethodChannel`, especificando el nombre del método y, opcionalmente, los argumentos.
2. **Plataforma (Host)**: El código nativo (Android o iOS) escucha en el mismo `MethodChannel`. Cuando recibe una llamada a un método, ejecuta la lógica nativa correspondiente y devuelve un resultado (o un error) a Dart.

Los datos se serializan y deserializan automáticamente entre Dart y la plataforma utilizando un formato estándar (generalmente JSON-like) que Flutter maneja internamente.

9.1.2 Ejemplo: Obtener la versión del sistema operativo

Vamos a crear un ejemplo simple para obtener la versión del sistema operativo (Android o iOS) utilizando un `MethodChannel`.

Paso 1: Código Dart (lib/main.dart)

```dart
import 'package:flutter/material.dart';
import 'package:flutter/services.dart'; // Importa para
MethodChannel

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

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Platform Channel Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const HomePage(),
    );
  }
}

class HomePage extends StatefulWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  State<HomePage> createState() => _HomePageState();
}
```

```dart
class _HomePageState extends State<HomePage> {
  static const platform = MethodChannel('com.example.app/
version'); // Define el canal
  String _version = 'Desconocida';

  Future<void> _getPlatformVersion() async {
    String version;
    try {
      // Invoca el método 'getPlatformVersion' en la plataforma
      final String result = await
platform.invokeMethod('getPlatformVersion');
      version = 'Versión de la plataforma: $result';
    } on PlatformException catch (e) {
      version = "Fallo al obtener la versión: '${e.message}'.";
    }

    setState(() {
      _version = version;
    });
  }

  @override
  void initState() {
    super.initState();
    _getPlatformVersion(); // Llama al método al iniciar la
página
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Platform Channel Demo'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(_version),
            ElevatedButton(
              onPressed: _getPlatformVersion,
              child: const Text('Obtener Versión de Nuevo'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Paso 2: Código Nativo Android (kotlin/MainActivity.kt)

En `android/app/src/main/kotlin/com/example/tu_app/MainActivity.kt`:

```
package com.example.tu_app

import androidx.annotation.NonNull
import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine
import io.flutter.plugin.common.MethodChannel

class MainActivity: FlutterActivity() {
  private val CHANNEL = "com.example.app/version"

  override fun configureFlutterEngine(@NonNull flutterEngine:
FlutterEngine) {
    super.configureFlutterEngine(flutterEngine)
    MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
CHANNEL).setMethodCallHandler {
      call, result ->
      // Este método es invocado desde Dart
      if (call.method == "getPlatformVersion") {
        val androidVersion = android.os.Build.VERSION.RELEASE
        result.success("Android $androidVersion")
      } else {
        result.notImplemented()
      }
    }
  }
}
```

Paso 3: Código Nativo iOS (swift/AppDelegate.swift)

En `ios/Runner/AppDelegate.swift`:

```
import UIKit
import Flutter

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    let controller : FlutterViewController =
window?.rootViewController as! FlutterViewController
    let versionChannel = FlutterMethodChannel(name:
"com.example.app/version",
```

```
                                                binaryMessenger:
controller.binaryMessenger)

    versionChannel.setMethodCallHandler({ (call:
FlutterMethodCall, result: @escaping FlutterResult) -> Void in
      // Este método es invocado desde Dart
      guard call.method == "getPlatformVersion" else {
        result(FlutterMethodNotImplemented)
        return
      }
      result("iOS " + UIDevice.current.systemVersion)
    })

    GeneratedPluginRegistrant.register(with: self)
    return super.application(application,
didFinishLaunchingWithOptions: launchOptions)
  }
}
```

Este ejemplo demuestra la comunicación básica. Puedes pasar argumentos de Dart a la plataforma y viceversa, y manejar diferentes tipos de datos. Para interacciones más complejas, como el acceso a hardware o la integración con SDKs de terceros, la lógica nativa será más elaborada.

9.2 Uso de paquetes de terceros para funcionalidades nativas

Aunque los Platform Channels te dan un control total sobre la comunicación nativa, escribir y mantener código nativo para cada plataforma puede ser tedioso y propenso a errores. Afortunadamente, la comunidad de Flutter ha creado una vasta colección de paquetes que abstraen la complejidad de los Platform Channels y te permiten acceder a funcionalidades nativas con solo unas pocas líneas de código Dart.

Estos paquetes son la forma preferida de integrar funcionalidades nativas en la mayoría de las aplicaciones Flutter, ya que te ahorran tiempo y esfuerzo, y suelen estar bien mantenidos y probados.

Cómo encontrar y usar paquetes:

1. **pub.dev**: El repositorio oficial de paquetes de Dart y Flutter (pub.dev) es el mejor lugar para buscar paquetes. Puedes buscar por funcionalidad (por ejemplo, "camera", "location", "bluetooth").
2. **Popularidad y Calidad**: Al elegir un paquete, considera su popularidad, la frecuencia de sus actualizaciones, la calidad de su documentación y si es un paquete "oficial" (mantenido por el equipo de Flutter o Google) o de la comunidad.

Ejemplos de paquetes populares para funcionalidades nativas:

- **`image_picker`**: Permite seleccionar imágenes o videos de la galería del dispositivo o tomar nuevas fotos/videos con la cámara.
    - **Uso**: Añade `image_picker` a `pubspec.yaml`, configura los permisos en `AndroidManifest.xml` (Android) y `Info.plist` (iOS), y luego usa `ImagePicker().pickImage()`.
- **`geolocator`**: Proporciona acceso a la ubicación GPS del dispositivo, incluyendo la ubicación actual, actualizaciones de ubicación y geocodificación.
    - **Uso**: Añade `geolocator` a `pubspec.yaml`, configura los permisos de ubicación en los archivos nativos, y luego usa `Geolocator.getCurrentPosition()` o `Geolocator.getPositionStream()`.
- **`url_launcher`**: Permite abrir URLs en el navegador web predeterminado, enviar correos electrónicos, hacer llamadas telefónicas o enviar SMS.
    - **Uso**: Añade `url_launcher` a `pubspec.yaml`, y luego usa `launchUrl(Uri.parse("https://flutter.dev"))`.
- **`battery_plus`**: Proporciona información sobre el estado de la batería del dispositivo (nivel, estado de carga).
- **`connectivity_plus`**: Permite verificar el estado de la conexión a internet del dispositivo.
- **`shared_preferences`**: (Ya cubierto en el Capítulo 5) Para almacenar datos simples de clave-valor, que internamente utiliza APIs nativas.
- **`sqflite`**: (Ya cubierto en el Capítulo 5) Para bases de datos SQLite locales, que interactúa con la implementación nativa de SQLite.

Ejemplo de uso de `image_picker`:

1. **Añadir la dependencia**: En tu `pubspec.yaml`: `yaml dependencies: flutter: sdk: flutter image_picker: ^1.0.7 # Usa la última versión estable` Luego, ejecuta `flutter pub get`.
2. **Configurar permisos (Android)**: En `android/app/src/main/AndroidManifest.xml`, añade los permisos necesarios dentro de la etiqueta `<manifest>`: `xml <uses-permission`

`android:name="android.permission.CAMERA" /> <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />`

3. **Configurar permisos (iOS)**: En `ios/Runner/Info.plist`, añade las siguientes claves con descripciones de uso (estas se mostrarán al usuario cuando la aplicación solicite acceso): `xml <key>NSPhotoLibraryUsageDescription</key> <string>Esta aplicación necesita acceso a tu galería de fotos para seleccionar imágenes.</string> <key>NSCameraUsageDescription</key> <string>Esta aplicación necesita acceso a tu cámara para tomar fotos.</string> <key>NSMicrophoneUsageDescription</key> <string>Esta aplicación necesita acceso a tu micrófono para grabar videos.</string>`

4. **Código Dart para usar `image_picker`:**

```dart import 'package:flutter/material.dart'; import 'package:image_picker/image_picker.dart'; import 'dart:io'; // Para File

class ImagePickerPage extends StatefulWidget { const ImagePickerPage({Key? key}) : super(key: key);

@override State createState() => _ImagePickerPageState(); }

class _ImagePickerPageState extends State { File? _imageFile; final ImagePicker _picker = ImagePicker();

Future _pickImage(ImageSource source) async { final pickedFile = await _picker.pickImage(source: source);

  setState(() {
    if (pickedFile != null) {
      _imageFile = File(pickedFile.path);
    } else {
      print('No se seleccionó ninguna imagen.');
    }
  });

}

@override Widget build(BuildContext context) { return Scaffold( appBar: AppBar( title: const Text('Seleccionar Imagen'), ), body: Center( child: Column( mainAxisAlignment: MainAxisAlignment.center, children: [ _imageFile == null ? const Text('No hay imagen seleccionada.') : Image.file(_imageFile!, height: 200), const SizedBox(height: 20), ElevatedButton( onPressed: () =>
```

_pickImage(ImageSource.camera), child: const Text('Tomar Foto con Cámara'),), ElevatedButton(onPressed: () => _pickImage(ImageSource.gallery), child: const Text('Seleccionar de Galería'),),],),),); } } ```

Al utilizar paquetes de terceros, puedes aprovechar el trabajo de la comunidad y acceder a una amplia gama de funcionalidades nativas de manera eficiente y con menos código.

Capítulo 10: Temas Avanzados y Buenas Prácticas

Este capítulo final profundiza en algunos temas más avanzados y consolida una serie de buenas prácticas que te ayudarán a construir aplicaciones Flutter de alta calidad, escalables y mantenibles. Desde la internacionalización hasta la accesibilidad y la integración continua, estos temas son cruciales para llevar tus habilidades de desarrollo Flutter al siguiente nivel.

10.1 Internacionalización y localización

Para que tu aplicación sea accesible a una audiencia global, es fundamental soportar múltiples idiomas y adaptarla a las convenciones culturales de diferentes regiones. Este proceso se conoce como internacionalización (i18n) y localización (l10n).

- **Internacionalización (i18n)**: El proceso de diseñar y desarrollar una aplicación de tal manera que pueda ser adaptada a diferentes idiomas y regiones sin cambios de ingeniería.
- **Localización (l10n)**: El proceso de adaptar una aplicación internacionalizada para una región o idioma específico, incluyendo la traducción de textos, el formato de fechas y números, y la adaptación de recursos gráficos.

Flutter proporciona un soporte robusto para la internacionalización y localización a través del paquete `flutter_localizations` y la clase `AppLocalizations`.

Pasos para implementar i18n y l10n:

1. **Añadir dependencias**: En tu `pubspec.yaml`: ```yaml dependencies: flutter: sdk: flutter flutter_localizations: sdk: flutter # Necesario para las delegaciones de localización intl: ^0.18.1 # Para herramientas de internacionalización

   dev_dependencies: intl_translation: ^0.17.11 # Para generar archivos de traducción `` Luego, ejecuta` flutter pub get`.

2. **Configurar `MaterialApp`**: En tu `MaterialApp`, especifica los `localizationsDelegates` y `supportedLocales`.

```dart import 'package:flutter/material.dart'; import 'package:flutter_localizations/flutter_localizations.dart'; import 'package:tu_app/l10n/app_localizations.dart'; // Archivo generado

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

class MyApp extends StatelessWidget { const MyApp({Key? key}) : super(key: key);

@override Widget build(BuildContext context) { return MaterialApp( title: 'Demo de Internacionalización', localizationsDelegates: const [ AppLocalizations.delegate, // Tu delegado personalizado GlobalMaterialLocalizations.delegate, GlobalWidgetsLocalizations.delegate, GlobalCupertinoLocalizations.delegate, ], supportedLocales: const [ Locale('en', ''), // Inglés Locale('es', ''), // Español Locale('fr', ''), // Francés ], home: const HomePage(), ); } } ```

3. **Crear archivos ARB (Application Resource Bundle)**: Estos archivos JSON contienen las cadenas de texto para cada idioma. Crea un directorio `l10n` en la raíz de tu proyecto y dentro de él, archivos como `app_en.arb`, `app_es.arb`, etc.

   `l10n/app_en.arb`: ```json {

"@@locale": "en", "helloWorld": "Hello World!", "welcomeMessage": "Welcome, {name}!", "counterText": "You have pushed the button this many times: {count}" } ```

````
`l10n/app_es.arb`:
```json
{
  "@@locale": "es",
  "helloWorld": "¡Hola Mundo!",
  "welcomeMessage": "¡Bienvenido, {name}!",
  "counterText": "Has presionado el botón esta cantidad de veces: {count}"
}
```

Las entradas con `@{}` (como `welcomeMessage` y `counterText`) indican que la cadena tiene parámetros.
````

1. **Generar clases de localización**: Ejecuta el siguiente comando en tu terminal para generar el código Dart a partir de los archivos ARB: `bash flutter gen-l10n` Esto creará un archivo `app_localizations.dart` (y otros) en el directorio `.dart_tool/flutter_gen/gen_l10n/` que contiene las clases `AppLocalizations` y `AppLocalizationsDelegate`.
````

2. **Usar las cadenas localizadas en tu código**: Accede a las cadenas usando `AppLocalizations.of(context)`.

   ```dart import 'package:flutter/material.dart'; import 'package:tu_app/l10n/app_localizations.dart';

   class HomePage extends StatelessWidget { const HomePage({Key? key}) : super(key: key);

   @override Widget build(BuildContext context) { return Scaffold( appBar: AppBar( title: Text(AppLocalizations.of(context)!.helloWorld), ), body: Center( child: Column( mainAxisAlignment: MainAxisAlignment.center, children: [ Text( AppLocalizations.of(context)!.welcomeMessage("Juan"), style: const TextStyle(fontSize: 24), ), Text( AppLocalizations.of(context)!.counterText(5), style: const TextStyle(fontSize: 24), ), ], ), ), ); } } ```

La internacionalización es un paso crucial para llegar a una audiencia global y ofrecer una experiencia de usuario inclusiva.

## 10.2 Accesibilidad

La accesibilidad en el desarrollo de aplicaciones se refiere a la práctica de diseñar y construir aplicaciones que puedan ser utilizadas por personas con diversas capacidades, incluyendo aquellas con discapacidades visuales, auditivas, motoras o cognitivas. Flutter ofrece herramientas y widgets que facilitan la creación de aplicaciones accesibles.

**Principios clave de accesibilidad en Flutter:**

- **Etiquetas semánticas (`Semantics`)**: Flutter utiliza un árbol semántico para describir la UI a los servicios de accesibilidad (como lectores de pantalla). Asegúrate de que todos los elementos interactivos y de contenido tengan etiquetas semánticas claras y descriptivas.
    - Muchos widgets de Material Design y Cupertino ya proporcionan semántica por defecto. Sin embargo, para widgets personalizados o combinaciones complejas, es posible que necesites envolverlos en un widget `Semantics`. `dart Semantics( label: "Botón para añadir un nuevo elemento", child: FloatingActionButton( onPressed: () {}, child: Icon(Icons.add), ), );`
   ```

- **Contraste de color**: Asegúrate de que haya suficiente contraste entre el texto y el fondo para que sea legible para personas con baja visión o daltonismo. Las directrices de Material Design y Cupertino suelen seguir estándares de contraste.
- **Tamaño de fuente y escalabilidad**: Permite que los usuarios ajusten el tamaño de la fuente de su dispositivo. Flutter respeta la configuración de escala de texto del sistema operativo por defecto. Evita usar tamaños de fuente fijos que no escalen.
- **Navegación por teclado y enfoque**: Asegúrate de que todos los elementos interactivos sean accesibles y navegables mediante teclado o dispositivos de entrada alternativos. El widget `FocusNode` y `FocusTraversalGroup` son útiles para gestionar el enfoque.
- **Alternativas de texto para imágenes**: Proporciona descripciones de texto (`alt text`) para todas las imágenes y elementos visuales que transmitan información. Esto es crucial para los usuarios de lectores de pantalla.
- **Animaciones y movimiento**: Ofrece opciones para reducir o deshabilitar animaciones para usuarios sensibles al movimiento. Considera usar `MediaQuery.of(context).disableAnimations`.
- **Pruebas de accesibilidad**: Utiliza herramientas de accesibilidad del sistema operativo (como TalkBack en Android o VoiceOver en iOS) para probar tu aplicación y asegurarte de que es utilizable para todos.

La accesibilidad no es solo una cuestión de cumplimiento, sino una parte fundamental de la creación de aplicaciones inclusivas que puedan ser disfrutadas por la mayor cantidad de personas posible.

10.3 Patrones de diseño en Flutter

Adoptar patrones de diseño probados es esencial para construir aplicaciones Flutter escalables, mantenibles y fáciles de entender. Si bien Flutter no impone un patrón específico, la comunidad ha adoptado varios que se adaptan bien a su naturaleza declarativa y reactiva.

10.3.1 Patrón MVVM (Model-View-ViewModel)

El patrón MVVM es muy popular en Flutter, especialmente cuando se combina con soluciones de gestión de estado como `provider` o `GetX`. Separa la aplicación en tres componentes principales:

- **Model**: Representa los datos y la lógica de negocio. Es independiente de la UI.

- **View**: La interfaz de usuario (widgets de Flutter). Es pasiva y solo muestra los datos que recibe del ViewModel y envía las interacciones del usuario al ViewModel.
- **ViewModel**: Actúa como un intermediario entre el Model y la View. Contiene la lógica de presentación, expone los datos del Model a la View de una manera que la View pueda consumir fácilmente, y maneja las interacciones del usuario, actualizando el Model según sea necesario.

Ventajas:

- **Separación de preocupaciones**: Clara distinción entre UI, lógica de presentación y lógica de negocio.
- **Probabilidad**: Los ViewModels son fáciles de probar de forma unitaria, ya que no tienen dependencias directas de la UI.
- **Reutilización**: Los ViewModels pueden ser reutilizados en diferentes Views.

10.3.2 Patrón BLoC (Business Logic Component)

Ya cubierto en el Capítulo 3, BLoC es un patrón de arquitectura que utiliza streams para gestionar el estado de la aplicación. Se centra en la separación de la lógica de negocio de la UI, utilizando eventos como entrada y estados como salida.

Ventajas:

- **Previsibilidad**: El flujo de datos unidireccional y el uso de eventos y estados inmutables hacen que el estado de la aplicación sea predecible.
- **Probabilidad**: Los BLoCs son altamente probables, ya que son funciones puras que transforman eventos en estados.
- **Escalabilidad**: Adecuado para aplicaciones grandes y complejas con lógica de negocio intrincada.

10.3.3 Patrón Provider (con ChangeNotifier)

También cubierto en el Capítulo 3, el patrón Provider, especialmente cuando se usa con `ChangeNotifier`, es una forma ligera y eficiente de gestionar el estado. Aunque no es un patrón de arquitectura completo como MVVM o BLoC, se puede usar para implementar la separación de preocupaciones.

Ventajas:

- **Simplicidad**: Fácil de aprender y usar para la mayoría de los casos de uso.
- **Rendimiento**: Optimizado para reconstrucciones eficientes de widgets.
- **Flexibilidad**: Se puede combinar con otros patrones o enfoques.

10.3.4 Otros patrones y consideraciones

- **Repository Pattern**: Abstrae la fuente de datos (API, base de datos local) de la lógica de negocio. Tu ViewModel o BLoC interactuaría con un repositorio, sin preocuparse de dónde provienen los datos.
- **Service Locator / Dependency Injection**: Para gestionar las dependencias de tu aplicación. Paquetes como `get_it` o el propio sistema de inyección de dependencias de `GetX` pueden ser útiles.
- **Clean Architecture**: Un enfoque más amplio que organiza el código en capas (dominio, datos, presentación) para lograr una alta separación de preocupaciones y facilitar la prueba y el mantenimiento.

La elección del patrón de diseño dependerá de la complejidad de tu aplicación, el tamaño de tu equipo y tus preferencias personales. Lo más importante es elegir un patrón y ser consistente en su aplicación en todo el proyecto.

10.4 CI/CD para aplicaciones Flutter (introducción)

La Integración Continua (CI) y el Despliegue Continuo (CD) son prácticas de desarrollo de software que automatizan los procesos de construcción, prueba y despliegue de aplicaciones. Implementar CI/CD para tus aplicaciones Flutter puede mejorar significativamente la calidad del software, reducir errores y acelerar el ciclo de lanzamiento.

10.4.1 Integración Continua (CI)

La CI implica fusionar los cambios de código en un repositorio central con frecuencia (varias veces al día) y ejecutar automáticamente pruebas y compilaciones para detectar errores tempranamente. Para Flutter, esto significa:

- **Automatización de pruebas**: Cada vez que se sube código al repositorio, se ejecutan automáticamente las pruebas unitarias, de widgets y de integración.
- **Análisis de código estático**: Herramientas como `flutter analyze` o `dart analyze` se ejecutan para identificar problemas de estilo, errores potenciales y malas prácticas.
- **Construcción de artefactos**: Se generan automáticamente los APKs, AABs, IPAs o paquetes web/desktop.

Herramientas populares de CI para Flutter:

- **GitHub Actions**: Integrado directamente con GitHub, permite definir flujos de trabajo de CI/CD en archivos YAML.
- **GitLab CI/CD**: Similar a GitHub Actions, pero para repositorios GitLab.

- **Bitrise**: Una plataforma de CI/CD especializada en aplicaciones móviles, con soporte nativo para Flutter.
- **Codemagic**: Otra plataforma de CI/CD diseñada específicamente para Flutter, que simplifica la configuración y el despliegue.

10.4.2 Despliegue Continuo (CD)

El CD extiende la CI al automatizar el proceso de despliegue de la aplicación a entornos de prueba o directamente a las tiendas de aplicaciones (Google Play Store, Apple App Store). Esto puede incluir:

- **Despliegue automático a entornos de prueba**: Después de pasar las pruebas de CI, la aplicación se despliega automáticamente a un entorno de pruebas internas o a un grupo de testers.
- **Despliegue a tiendas de aplicaciones**: Una vez que la aplicación ha sido aprobada en las pruebas, se puede configurar para que se despliegue automáticamente a las tiendas de aplicaciones.

Consideraciones para CD en Flutter:

- **Firmado de la aplicación**: Las claves de firma deben gestionarse de forma segura en el entorno de CI/CD (por ejemplo, como variables de entorno cifradas).
- **Metadatos de la tienda**: La información de la ficha de la tienda (descripciones, capturas de pantalla) puede gestionarse a través de herramientas como `fastlane` para automatizar las actualizaciones.
- **Control de versiones**: Asegúrate de que tu proceso de CI/CD actualice automáticamente el número de versión y de compilación de tu aplicación.

Ejemplo conceptual de GitHub Actions para Flutter:

```yaml
# .github/workflows/flutter_ci.yaml
name: Flutter CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - uses: subosito/flutter-action@v2
        with:
          channel: 'stable'

      - name: Install dependencies
```

```
        run: flutter pub get

      - name: Analyze code
        run: flutter analyze

      - name: Run tests
        run: flutter test

      - name: Build Android App Bundle
        run: flutter build appbundle --release

      - name: Build iOS Archive (on macOS runner)
        if: runner.os == 'macOS'
        run: flutter build ios --release --no-codesign

      # Puedes añadir pasos para subir a Firebase App
Distribution, Google Play, App Store Connect, etc.
```

Implementar CI/CD puede parecer una inversión de tiempo al principio, pero los beneficios a largo plazo en términos de calidad, velocidad y fiabilidad del desarrollo son inmensos. Es una práctica esencial para cualquier equipo de desarrollo serio.