Introducción a Flutter

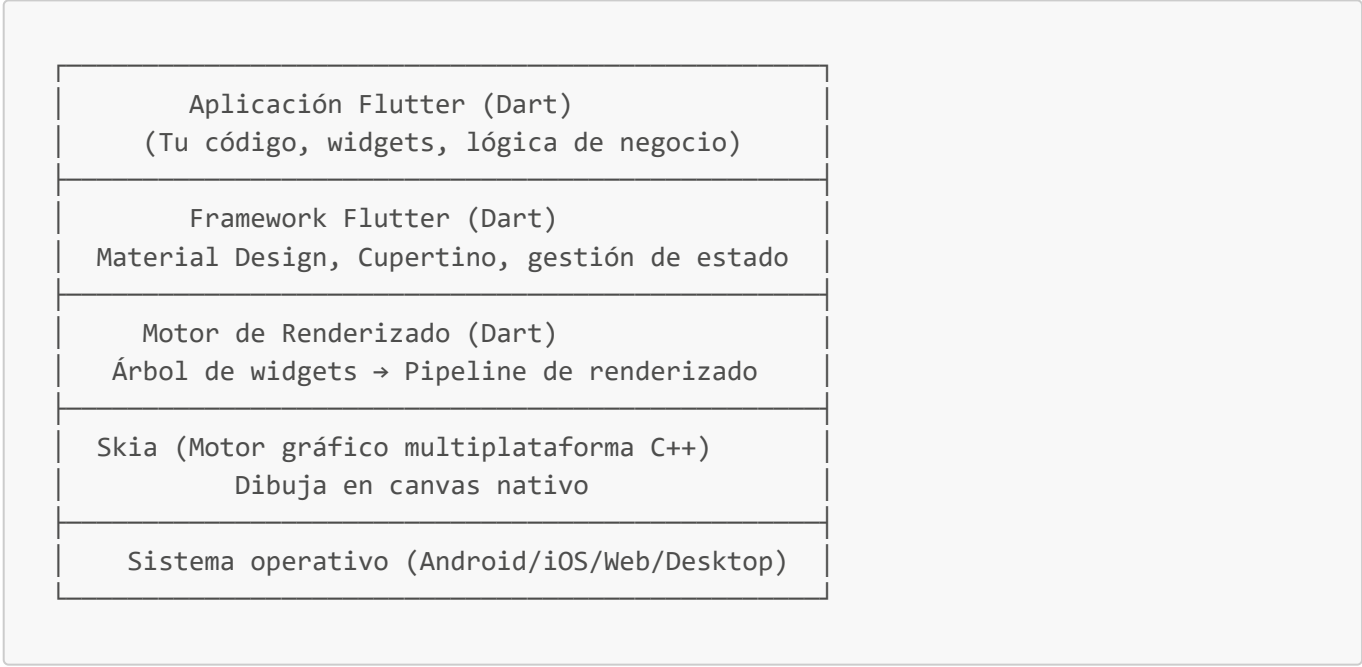
¿Qué es Flutter?

Flutter es un framework de código abierto desarrollado por **Google** para crear aplicaciones móviles nativas multiplataforma de alta calidad. Permite desarrollar aplicaciones para **Android, iOS, Web y escritorio** (Windows, macOS, Linux) desde una única base de código.

Características Principales

- **Multiplataforma**: Escribe una vez, ejecuta en múltiples plataformas
- **Alto rendimiento**: Compilación a código nativo para máximo desempeño
- **Hot Reload**: Recarga en caliente para desarrollo rápido e iterativo
- **Widgets ricos**: Amplio catálogo de componentes de UI personalizables
- **Material Design e iOS Design**: Soporte integrado para ambos estilos
- **Desarrollo rápido**: Reduce tiempos de desarrollo significativamente
- **Comunidad activa**: Amplio ecosistema de paquetes y librerías

Arquitectura de Flutter



Motor de Renderizado (Dart)

El **motor de renderizado** es la capa escrita en **Dart** que transforma tu código Flutter en píxeles en la pantalla. Es el corazón de Flutter y realiza el trabajo pesado de:

Funciones Principales:

1. **Construcción del árbol de widgets**
    - Procesa tus widgets (StatelessWidget, StatefulWidget, etc.)
    - Crea un árbol jerárquico de componentes

- Ejemplo: `MaterialApp → Scaffold → AppBar → Text`

2. **Detección de cambios**
   - Monitorea cambios de estado con `setState()`
   - Identifica qué widgets han cambiado
   - Solo rebuildea lo necesario (eficiente)
3. **Layout y posicionamiento**
   - Calcula el tamaño y posición de cada widget
   - Respeta restricciones (constraints)
   - Sistema de box constraints similar a CSS Flexbox
4. **Renderizado selectivo**
   - Genera comandos de renderizado
   - Optimiza para solo dibujar lo visible
   - Maneja transformaciones y animaciones

Ejemplo del flujo:

```
Tu código Dart
    ↓
build() se ejecuta
    ↓
Árbol de widgets (Widget Tree)
    ↓
Motor detecta cambios (Dirty check)
    ↓
Rebuild selectivo
    ↓
Cálculos de layout
    ↓
Comandos para Skia
    ↓
Pantalla actualizada ✨
```

Skia (Motor Gráfico Nativo)

Skia es un motor gráfico de código abierto desarrollado por **Google** en **C++**. Es la capa que **realmente dibuja** los píxeles en la pantalla.

¿Por qué Skia y no el sistema operativo nativo?

Aspecto	Sistema Nativo	Skia
Consistencia	Diferentes en cada SO	Igual en todas partes
Control	Limitado	Completo

Aspecto	Sistema Nativo	Skia
Rendimiento	Variable	Optimizado
Customización	Limitada	Total

Funciones de Skia:

1. **Renderizado de gráficos**
    - Dibuja formas (rectángulos, círculos, paths)
    - Aplica colores, gradientes, patrones
    - Renderiza texto con fuentes nativas
2. **Composición y capas**
    - Maneja capas de composición
    - Aplica efectos como sombras y blur
    - Optimiza el renderizado con VSync (sincronización vertical)
3. **Hardware acceleration**
    - Usa GPU cuando está disponible
    - Rasterización eficiente
    - Aprovecha instrucciones nativas del SO

¿Qué es un Wrapper?

Un **Wrapper** es una **capa de adaptación** que envuelve código o funcionalidad existente para proporcionar una interfaz uniforme y consistente. Es como una "caja de traducción" entre diferentes plataformas.

Analogía del mundo real:

```
Enchufe USA (110V)  →  [ADAPTADOR]  →  Enchufe EU (220V)
  (Código nativo)       (Wrapper)        (Tu código Dart)

El adaptador no cambia los enchufes, simplemente
permite que funcionen juntos sin problemas.
```

En Flutter, Skia usa Wrappers así:

```
┌─────────────────────────────────────────┐
│         Tu código Flutter (Dart)        │
│   (Mismo para todas las plataformas)    │
└─────────────────┬───────────────────────┘
                  ↓
        Llamadas a Skia (API uniforme)
                  ↓
     ┌────────────┬────────────┬────────────┐
```

```
      ↓           ↓           ↓           ↓
  ┌──────┐   ┌──────┐   ┌──────┐   ┌──────┐
  │ Android│  │ iOS  │   │ Web  │   │Desktop │
  │Wrapper│   │Wrapper│  │Wrapper│  │Wrapper │
  │OpenGL │   │Metal │   │Canvas │   │Vulkan  │
  └──────┘   └──────┘   └──────┘   └──────┘
     ↓           ↓           ↓           ↓
  GPU Native   GPU Native   Canvas      GPU Native
  Android      iOS          HTML5       Windows/Mac/Linux
```

¿Por qué necesita Wrappers?

Cada SO tiene su propia forma de dibujar gráficos:

Plataforma	API Nativa	Wrapper de Skia
Android	OpenGL / Vulkan	Convierte comandos Skia a OpenGL/Vulkan
iOS	Metal	Convierte comandos Skia a Metal
Web	Canvas HTML5 / WebGL	Convierte comandos Skia a Canvas/WebGL
Windows	DirectX / OpenGL	Convierte comandos Skia a Vulkan
macOS	Metal	Convierte comandos Skia a Metal
Linux	Vulkan / OpenGL	Convierte comandos Skia a Vulkan/OpenGL

Ejemplo técnico:

```
// Tu código Dart (igual en todas partes)
Container(
  color: Colors.blue,
  width: 100,
  height: 100,
)

// El wrapper traduce esto así:

// EN ANDROID:
// OpenGL.drawRect(x, y, 100, 100, Color.BLUE)

// EN iOS:
// Metal.renderBuffer(createBlueRect(100, 100))

// EN WEB:
// Canvas.fillRect(x, y, 100, 100, '#0000FF')

// Resultado visual: Exactamente lo mismo en todas partes ✨
```

Flujo completo con Wrapper:

```
1. Tu código Flutter:
   return Container(color: Colors.red, child: Text('Hola'));

2. Motor de renderizado:
   Crea árbol de widgets y calcula layout

3. Skia genera comandos:
   "Dibuja rectángulo rojo"
   "Dibuja texto 'Hola' en blanco"

4. WRAPPER (la magia ocurre aquí):
   Android wrapper: →  OpenGL.drawRect(); OpenGL.drawText()
   iOS wrapper:    →  Metal.encode(); Metal.commit()
   Web wrapper:    →  ctx.fillRect(); ctx.fillText()

5. SO nativo:
   Android: Envía a GPU Android
   iOS: Envía a GPU iOS (Metal)
   Web: Envía a GPU del navegador

6. Resultado:
   ✅ Android: Se ve igual
   ✅ iOS: Se ve igual
   ✅ Web: Se ve igual
   ✅ Desktop: Se ve igual
```

Ventaja clave de los Wrappers en Flutter:

```
SIN Wrappers (escribir nativo):
┌──────────────────┐
│ iOS native code  │  ≠  ┌──────────────────┐
└──────────────────┘     │ Android code     │
                         └──────────────────┘
Mismo app → 2× código, 2× bugs, 2× mantenimiento ❌

CON Wrappers (Flutter):
┌──────────────────────────────────┐
│   Tu código Dart (UNA VEZ)       │
│                                  │
│  Wrappers automáticos manejan    │
│  iOS, Android, Web, Desktop      │
└──────────────────────────────────┘
"Write once, run everywhere" ✅
```

Wrappers en plugins Flutter:

Cuando usas un plugin (como `camera`, `location`, `firebase`), también usa wrappers:

```
// Tu código Dart
final image = await camera.takePicture();

// Plugin Flutter (capa Dart)
  ↓
// Wrapper Android (Kotlin/Java)
  → Camera nativa Android

// Wrapper iOS (Swift/Objective-C)
  → Camera nativa iOS

// Resultado: Ambas plataformas funcionan con el mismo código 📱
```

Tipos de Wrappers en Flutter:

```
1. WRAPPERS VISUALES (Skia)
   Convierten comandos gráficos a APIs nativas

2. WRAPPERS FUNCIONALES (Plugins)
   Convierten llamadas Dart a funcionalidad nativa

3. WRAPPERS DE PLATAFORMA
   Permiten comunicación entre Dart ↔ Código nativo
```

---

4. **Independencia de plataforma**
   - Mismo código visual en Android, iOS, Web
   - Wrapper específico por plataforma
   - En Android: OpenGL/Vulkan
   - En iOS: Metal
   - En Web: Canvas HTML5 o WebGL

Flujo de Skia:

```
Comandos de renderizado (del motor Dart)
    ↓
Skia procesa cada comando
    ↓
Convierte a instrucciones gráficas
    ↓
GPU/CPU (dependiendo de disponibilidad)
    ↓
Framebuffer del SO
    ↓
Pantalla del usuario ✨
```

¿Qué es VSync? 🔄

VSync significa **Vertical Synchronization** (Sincronización Vertical). Es un mecanismo crítico que asegura que Flutter renderice en sincronía perfecta con la pantalla de tu dispositivo.

¿Por qué es importante?

Cuando tu pantalla se actualiza (típicamente 60 veces por segundo en 60 Hz), ocurre de arriba a abajo:

```
Pantalla (60 Hz) - 16.67ms por frame
├─ 0ms ──→ Línea 0
├─ 8.3ms ─→ Línea 540 (pantalla a mitad)
├─ 16.67ms → Línea 1080 (pantalla completa)
└─ REPITE...

Si Flutter renderiza en el medio (8.3ms):
¡TEARING! 💔 (Pantalla rasgada)
```

Cómo VSync resuelve esto:

```
Flutter espera a que la pantalla esté lista ⏳
    ↓
Pantalla envía VSync SIGNAL (cada 16.67ms)
    ↓
Skia renderiza inmediatamente
    ↓
Nuevo frame listo para la siguiente línea 0
    ↓
Resultado: Animaciones SUAVE sin rasgaduras ✨
```

Comparación visual:

SIN VSync (MALO):

```
Tiempo →
Frame 1 renderizado:  [====]
                          ↓ Pantalla dibuja aquí (rasgada)
Pantalla mostrada:     [=========]
Frame 2 renderizado:         [====]
                             ↓ Pantalla dibuja aquí (rasgada)
Pantalla mostrada:     [=========]
```

CON VSync (BIEN):

```
Tiempo →
Frame 1 renderizado: [====] (listo)
Pantalla mostrada:   [====] (perfecto)
                     ↑ VSync Signal
Frame 2 renderizado:      [====] (listo)
Pantalla mostrada:        [====] (perfecto)
                          ↑ VSync Signal
```

En Flutter:

Flutter **automáticamente usa VSync** a través de Skia:

1. **En Android**: USA OpenGL/Vulkan con VSync
2. **En iOS**: USA Metal con sincronización nativa
3. **En Web**: Usa `requestAnimationFrame()` (equivalente a VSync)
4. **En Desktop**: Maneja VSync del SO

Velocidad de fotogramas (FPS):

```
// El objetivo de Flutter es 60 FPS (en pantallas de 60Hz)
// 1 segundo / 60 fotogramas = 16.67 ms por frame

// Desglose de tiempo en un frame:
// ├─ 0-10ms: Código Dart (build, setState, etc.)
// ├─ 10-15ms: Renderizado Skia
// └─ 15-16.67ms: Esperar VSync
// Total: ~16.67ms ✅

// Si Flutter tarda > 16.67ms en un frame:
// Result: FRAME DROP ❌ (vuelve a 30 FPS)
```

En dispositivos de 120 Hz:

Los nuevos smartphones tienen pantallas de 120Hz (8.33ms por frame):

```
// Para aprovechar 120 FPS:
// ├─ 0-5ms: Código Dart
// ├─ 5-7ms: Renderizado Skia
// └─ 7-8.33ms: VSync
// Muy apretado, pero Flutter puede hacerlo ⚡
```

Checklist de rendimiento con VSync:

- ✅ Mantén `build()` simple y rápido
- ✅ Evita operaciones pesadas en `build()`

- ✅ Usa `const` para widgets que no cambian
- ✅ Usa `SingleChildScrollView` o `ListView` para listas largas
- ❌ No hagas cálculos complejos en `build()`
- ❌ No llames APIs en `build()`
- ❌ No modifiques estado en `build()`

Debugging de FPS:

```
# Ver FPS en tiempo real
flutter run

# Activar overlay de performance (presiona 'P' en la terminal)
# O en código:
WidgetsApp.debugShowWidgetInspectorOverride = true;

# En Android Studio: Tools → Dart DevTools → Performance
```

Por qué es importante esta arquitectura

```
// Cuando escribes un widget como este:
class MiBoton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Text('Hola'),
    );
  }
}

// DETRÁS OCURRE:
// 1. Motor de renderizado (Dart):
//    - Convierte tu Widget en instrucciones de layout
//    - Calcula posición y tamaño
//
// 2. Skia (C++):
//    - Dibuja un rectángulo azul
//    - Renderiza el texto "Hola" encima
//
// RESULTADO: Lo ves en tu pantalla en 16ms (60 FPS)
```

Ventajas de esta separación

Ventaja	Beneficio
Código en Dart	Fácil de aprender y mantener

Ventaja	Beneficio
Motor en Dart	Lógica clara y debuggeable
Gráficos en Skia	Máximo rendimiento
Abstracción	No necesitas pensar en detalles de SO

Hot Reload y el Motor de Renderizado

El **Hot Reload** es posible gracias a cómo funciona el motor:

```
1. Detecta cambios en tu código (lib/main.dart)
2. Recompila solo la parte modificada
3. El motor de renderizado re-ejecuta build()
4. Solo los widgets afectados se redibujan
5. Todo ocurre en <100ms sin perder estado ⚡
```

Por eso Flutter es tan rápido para desarrollar: cambias código, presionas `r`, ¡y ves el resultado al instante!

Resumen Visual

```
┌─────────────────────────────────────┐
│   Motor Renderizado (Dart)          │
│   • Procesa widgets                 │
│   • Calcula layout                  │
│   • Detecta cambios                 │
│   • Genera comandos gráficos        │
└─────────────────────────────────────┘
              │
              ↓
       Comandos de dibujo
       (Dibuja rectángulo, texto...)
              ↓
┌─────────────▼───────────────────────┐
│   Skia (C++)                        │
│   • Dibuja en GPU/CPU               │
│   • Renderiza cada comando          │
│   • Optimiza performance            │
│   • Produce framebuffer             │
└─────────────────────────────────────┘
              │
              ↓
       Imagen final en pantalla 📱
```

Requisitos Previos

Para el desarrollo en Flutter necesitarás:

- **Dart SDK**: Se instala automáticamente con Flutter

- **Flutter SDK**: Descarga de https://flutter.dev
- **IDE recomendado**:
  - VS Code + extensión Flutter
  - Android Studio
  - IntelliJ IDEA
- **Emulador o dispositivo físico** para testing
- **Git**: Para control de versiones

Instalación de Flutter

1. Descargar Flutter SDK

```
# En Windows (PowerShell o CMD)
# Descarga desde: https://flutter.dev/docs/get-started/install/windows
# O usando git:
git clone https://github.com/flutter/flutter.git -b stable
```

2. Añadir Flutter al PATH

```
# En Windows, añade la ruta al PATH del sistema
# Ejemplo: C:\Users\tu_usuario\flutter\bin
```

3. Verificar la instalación

```
flutter doctor
```

Este comando verifica que todo esté correctamente instalado y muestra los requisitos pendientes.

Comandos Básicos de Flutter

Información y Diagnóstico

```
flutter doctor              # Verifica la instalación y dependencias
flutter doctor -v           # Versión detallada del diagnóstico
flutter --version           # Muestra versión de Flutter
dart --version              # Muestra versión de Dart
```

Crear Proyectos

```
# Crear nuevo proyecto
flutter create nombre_proyecto
```

```
# Crear proyecto con organización específica
flutter create --org com.example nombre_proyecto

# Crear proyecto específico para una plataforma
flutter create -t app nombre_proyecto          # App por defecto
flutter create -t package nombre_paquete       # Librería/Paquete
flutter create -t plugin nombre_plugin         # Plugin nativo
```

Ejecutar Aplicaciones

```
# Ejecutar en dispositivo/emulador conectado
flutter run

# Ejecutar con hot reload deshabilitado
flutter run --no-hot

# Ejecutar en navegador (Web)
flutter run -d chrome

# Listar dispositivos disponibles
flutter devices

# Ejecutar en dispositivo específico
flutter run -d device_id
```

Gestión de Dependencias

```
# Ver dependencias del proyecto
flutter pub dependencies

# Obtener dependencias
flutter pub get

# Actualizar dependencias
flutter pub upgrade

# Limpiar caché de pub
flutter pub cache clean

# Analizar dependencias
flutter pub outdated
```

Análisis y Testing

```
# Analizar código (lint)
flutter analyze
```

```
# Ejecutar tests
flutter test

# Ejecutar tests con coverage
# Coverage = porcentaje de código que está siendo probado por tests
# Genera reporte de cobertura de tests (qué % de tu código está cubierto)
flutter test --coverage

# Ejecutar tests específicos
flutter test test/widget_test.dart
```

Build y Release

```
# Build para Android
flutter build apk                    # Genera APK
flutter build appbundle              # Genera App Bundle para Play Store

# Build para iOS
flutter build ios

# Build para Web
flutter build web

# Build para escritorio (Windows)
flutter build windows

# Build para escritorio (macOS)
flutter build macos

# Build para escritorio (Linux)
flutter build linux
```

Otros Comandos Útiles

```
# Limpiar archivos generados
# Elimina carpetas build/, .dart_tool/ y caché de compilación
# Útil para resolver problemas de compilación o liberar espacio
flutter clean

# Formatea automáticamente el código Dart
# Aplica estándares de indentación y estilo de código
# Ayuda a mantener consistencia en el proyecto
flutter format lib/

# Crear launcher icons
# Genera los iconos de la app automáticamente para Android e iOS
# Requiere un plugin y archivo de configuración previo
flutter pub run flutter_launcher_icons:main
```

```
# Ver logs de la aplicación en tiempo real
# Muestra mensajes de debug, errores y warnings
# Muy útil para debugging de problemas en ejecución
flutter logs

# Detener los procesos de Flutter en ejecución
# Mata todos los procesos flutter (emulador, servidor, etc.)
# Útil cuando algo queda colgado o necesitas limpiar puertos
flutter kill
```

Estructura de un Proyecto Flutter

```
mi_proyecto/
├── android/                # Código nativo Android
├── build/                  # Archivos compilados (ignorar)
├── ios/                    # Código nativo iOS
├── lib/                    # 🎯 Código Dart principal
│   ├── main.dart           # Punto de entrada
│   ├── screens/            # Pantallas de la app
│   ├── widgets/            # Widgets personalizados
│   ├── models/             # Modelos de datos
│   ├── services/           # Servicios (API, BD, etc.)
│   └── utils/              # Utilidades y helpers
├── test/                   # Tests unitarios
├── pubspec.yaml            # Dependencias y configuración
├── pubspec.lock            # Lock file de dependencias
├── .gitignore              # Archivos a ignorar en git
└── README.md               # Documentación del proyecto
```

Ciclo de Vida de una App Flutter

```
main()
  ↓
runApp(MyApp())
  ↓
MaterialApp / CupertinoApp
  ↓
home: MyHomePage()
  ↓
Estado inicial cargado
  ↓
Widget.build()
  ↓
Renderizado en pantalla
```

Conceptos Clave

Widgets

Un widget es la unidad básica de construcción en Flutter. Todo es un widget.

```
// Widget sin estado
class MiWidgetSinEstado extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Text('Hola Flutter'),
    );
  }
}

// Widget con estado
class MiWidgetConEstado extends StatefulWidget {
  @override
  State<MiWidgetConEstado> createState() => _MiWidgetConEstadoState();
}

class _MiWidgetConEstadoState extends State<MiWidgetConEstado> {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Text('Hola Flutter'),
    );
  }
}
```

Hot Reload

Permite ver cambios en el código casi instantáneamente sin perder el estado de la app durante el desarrollo.

BuildContext

Representa la ubicación de un widget en el árbol de widgets. Se usa para acceder a temas, navegación, etc.

Diferencia: StatelessWidget vs StatefulWidget

StatelessWidget	**StatefulWidget**
Inmutable	Puede cambiar
No mantiene estado	Mantiene estado interno
Rendimiento mejor	Rendimiento similar
Ejemplo: botón estático	Ejemplo: checkbox

Primeros Pasos

1. Crear tu primer proyecto

```
flutter create hello_flutter
cd hello_flutter
flutter run
```

2. Editar `lib/main.dart`

```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: 'Mi App Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Bienvenido a Flutter')),
        body: const Center(
          child: Text('¡Hola Flutter!'),
        ),
      ),
    );
  }
}
```

3. Guardar y ver cambios con Hot Reload

- Presiona `r` en la terminal para hot reload
- Presiona `R` para hot restart (reinicio completo)

Recursos Útiles

- **Documentación oficial**: https://flutter.dev/docs
- **Dart Docs**: https://dart.dev/guides
- **Package pub.dev**: https://pub.dev
- **Flutter Community**: https://flutter.dev/community
- **Stack Overflow**: Tag `flutter` y `dart`
- **YouTube**: Flutter oficial channel

Próximos Temas

1. **Widgets Fundamentales**: Container, Row, Column, Stack, Scaffold
2. **Manejo de Estado**: setState, Provider, Riverpod, BLoC

3. **Navegación**: Navigator, rutas nombradas, go_router
4. **HTTP y APIs**: Llamadas REST, manejo de JSON, dio
5. **Persistencia de Datos**: SQLite, SharedPreferences, Hive
6. **Testing**: Tests unitarios y de widgets
7. **Deployment**: Publicar en App Store y Google Play

Última actualización: Enero 2026
Versión de Flutter recomendada: 3.x+