



Flutter

(una introducción)

José M. Torresano
Noviembre 2025
buzondelprofesor@gmail.com

Flutter

¿Qué es Flutter?

Es un kit de desarrollo de software (SDK) para interfaces de usuario (UI) de código abierto creado por Google en 2017, que permite desarrollar aplicaciones multiplataforma (móvil, web, de escritorio) a partir de una única base de código. Utiliza el lenguaje de programación Dart y ofrece un motor de gráficos propio (*Skia*) para un rendimiento rápido y una apariencia visual consistente en todos los dispositivos.

- **Stateful Hot Reload**, o recarga en caliente con estado, gracias a la compilación JIT(*just-in-time*), y que nos permite realizar cambios en el código y visualizarlos de manera instantánea en las vistas sin necesidad de volver a cargar todo el contexto gráfico.
- Rendimiento prácticamente nativo, ya que el código compilado de forma OAT(*ahead-of-time*) genera código nativo.
- Permite crear aplicaciones con interfaces gráficas llamativas, con un comportamiento prácticamente nativo.

¿Qué es Flutter?

De todas formas, debemos tener en cuenta que no estamos realizando desarrollos con un entorno puramente nativo, por lo que:

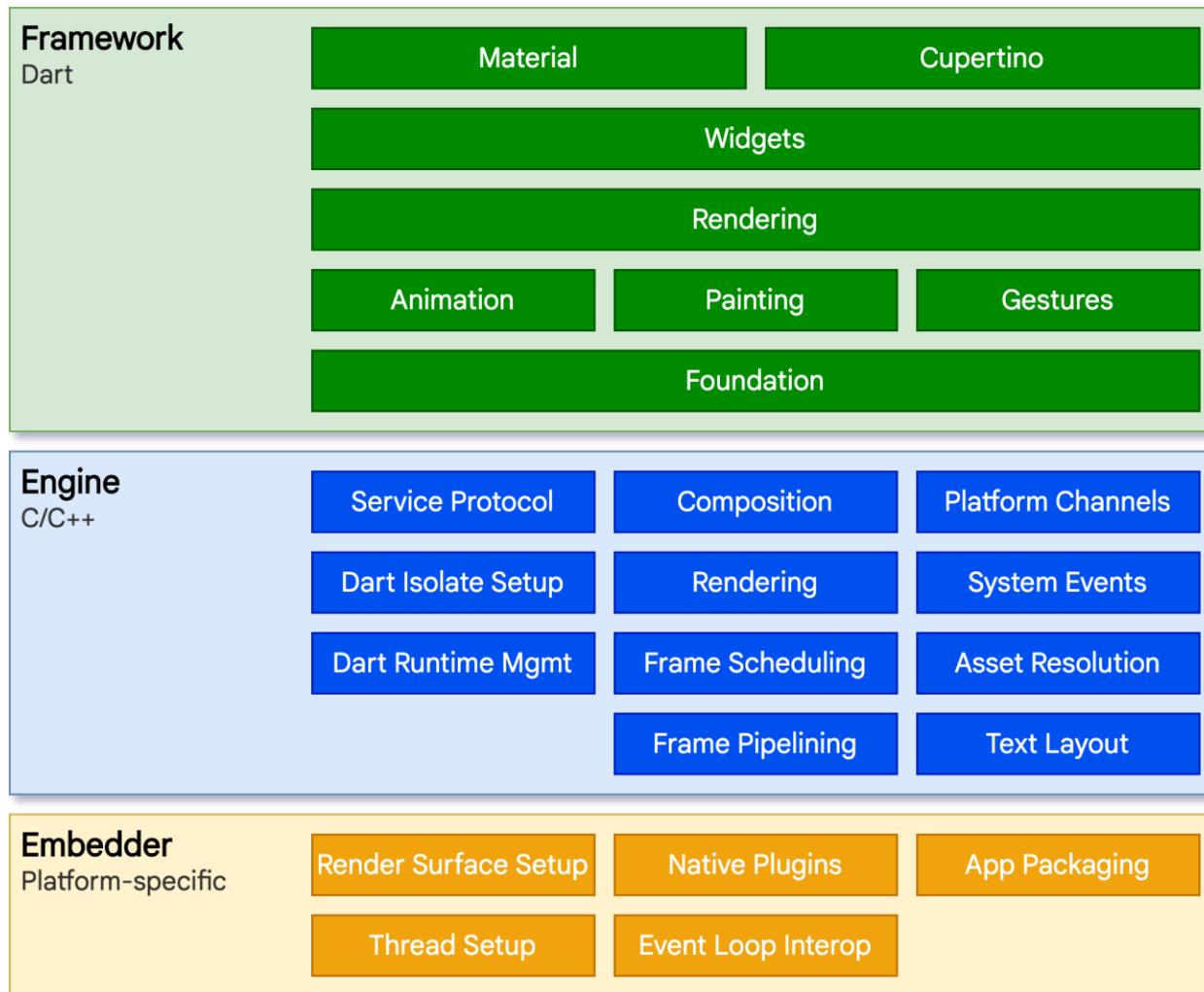
Principalmente, cuando desarrollemos para plataformas móviles, no dispondremos de forma inmediata de las últimas características de los sistemas operativos, ya que éstas se incorporan de forma prioritaria a los entornos nativos.

Las aplicaciones multiplataforma son más pesadas y menos escalables que las nativas, ya que deben estar preparadas para interactuar con distintos sistemas,

Tienen un rendimiento ligeramente inferior a las aplicaciones nativas puras, ya que no están tan optimizadas como las compiladas con las herramientas nativas.

Arquitectura de Flutter

La arquitectura de Flutter se compone de un sistema extensible de capas independientes, en donde cada capa superior depende de la capa inferior a ella.



Arquitectura de Flutter

Framework: La capa superior, desarrollada en Dart, nos proporciona un *framework* moderno y reactivo al lenguaje, y que consta de cuatro subcapas:

La capa superior, con las librerías Material y Cupertino, que implementan los controles en los diferentes lenguajes de diseño de Android y iOS,

La capa de widgets (*widgets*), que implementa de forma reactiva los diferentes tipos de controles y su composición.

La capa de representación, que gestiona el árbol de componentes y se encarga de su representación.

Las clases fundamentales básicas y las bibliotecas de animación, colores y renderizado.

Engine o motor de Flutter : Esta capa de soporte al *framework* está desarrollada en C y C++, y es la responsable del renderizado de cada *frame*. Este motor proporciona la implementación a bajo nivel de la API principal de Flutter: el motor gráfico Skia, la representación de texto, la gestión de la entrada y salida o el soporte a la accesibilidad, entre otros.

Embedder o integrador, encargado principalmente del empaquetado en formato nativo y la integración de la aplicación en el sistema operativo subyacente.

Instalación del SDK

Para realizar la instalación del SDK para Flutter, seguimos las instrucciones de la documentación oficial (<https://docs.flutter.dev/get-started/i>).

En primer lugar, accedemos a la web, y seleccionaremos si queremos una instalación rápida o a medida. En nuestro caso, elegiremos la rápida (que es la recomendada).

 Quick start
Recommended

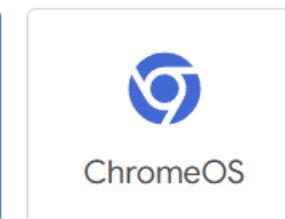
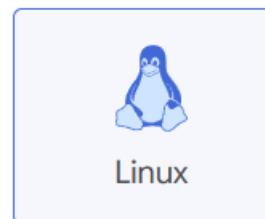
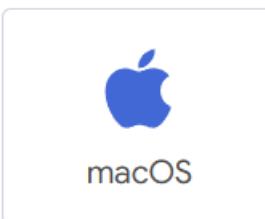
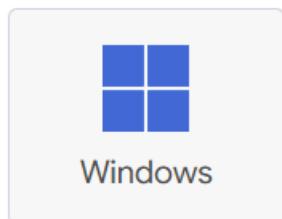
Use VS Code or another Code OSS-based editor to quickly install, set up, and try out Flutter development!

 Custom setup

Install the Flutter SDK, set up any initial target platform, and get started learning and developing with Flutter!

A continuación, seleccionaremos el sistema operativo en el que deseamos realizar la instalación. En nuestro caso, mostraremos los pasos realizados en Linux aunque comentaremos en paralelo la instalación en Windows.

If you'd like to follow the instructions for a different OS, please select one of the following.



Instalación del SDK

Descarga del software necesario

- ① Descarga e instala los paquetes necesarios.
- ② Descarga e instala Visual Studio Code

Instalar y configurar Flutter

- ① Iniciar VS Code
- ② Agrega la extensión de Flutter a VS Code
- ③ Instala Flutter con VS Code

Probar Flutter

Widgets

Interfaces declarativas vs imperativas

El desarrollo tradicional de interfaces de usuario se ha basado en un **estilo imperativo**, en el que definimos qué deseamos que se represente en la interfaz y cómo queremos hacerlo. Tanto en aplicaciones de escritorio, como web o móviles solemos especificar la interfaz de usuario mediante archivos basados en XML (XML puro, HTML, QML, etc.), que deben transformarse en elementos manipulables y enlazarlos de alguna manera con el código de la aplicación.

En contraposición a esta visión, el **paradigma declarativo**, propone un mayor nivel de abstracción, y se centra en el qué y no en el cómo . Aplicado al diseño de interfaces, la idea es que nos centramos en describir el estado actual de la interfaz, dejando para el framework la representación de éste y las transiciones entre estados. Es decir, la interfaz de usuario se genera en función del estado que queremos representar:

$$\text{UI} = f(\text{state})$$

La disposición en pantalla Tus métodos build El estado de la aplicación

Interfaces declarativas vs imperativas

Flutter es un framework declarativo. Ante cambios de estado, el árbol de widgets se reconstruye. No existe forma de referenciar un widget y cambiar sus propiedades directamente, sino que la interfaz de usuario se reconstruye a través de los métodos `build` de los widgets para representar al estado de la aplicación.

$$\text{UI} = f(\text{state})$$

La disposición en pantalla Tus métodos build El estado de la aplicación

Viendo la ecuación anterior, la mayor complejidad viene dada por como mantener el estado de la aplicación en los widgets y como aplicar modificaciones sobre el mismo, manteniendo sincronizada la interfaz.

Pensando de forma declarativa

Vemos un ejemplo simplificado. Imaginemos que queremos poner un texto de **Hola Mundo** centrado en pantalla. A grandes rasgos, con Android, por ejemplo, tendríamos en el archivo de descripción de la interfaz el siguiente elemento de tipo **TextView**:

```
<TextView  
    android:id="@+id/miTextView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

En el código de la aplicación, cuando inicializamos la interfaz, deberíamos buscar este **TextView**, haciendo uso del método **findViewById**, que devuelve un objeto de tipo **View** , que es quien representa los elementos gráficos, realizar un casting de esta vista a un **TextView** , y finalmente, acceder a la propiedad **text** del **TextView** y modificarla. Kotlin esto se haría con el siguiente código:

```
var miTextView=findViewById<TextView>(R.id.miTextView)  
miTextView.text="Hola Mundo"
```

En desarrollo web nos encontraríamos con la misma solución

```
<div id="Contenedor"><p></p></div>  
...  
var p=document.getElementById("Contenedor").firstElementChild;  
p.innerHTML="Hola Mundo";
```

Pensando de forma declarativa

El planteamiento de este problema de forma declarativa sería: Quiero un texto "Hola Mundo" centrado en pantalla . En código, esto se traduciría en algo parecido a lo siguiente:

```
Center(  
    child: Text(  
        'Hola Mundo'  
    )  
)
```

←
Paso de argumentos
con nombre

Con lo que definimos un contenedor que centra los elementos, y dentro del cual tenemos un texto **Hola Mundo**.

Si observamos el código veremos que estamos utilizando constructores con parámetros posicionales y por nombre de diferente forma:

- El constructor del elemento Center recibe un único argumento con nombre: `child`, que indica qué elemento tiene dentro.
- Este elemento interno es un Text, que recibe a su constructor como único argumento el texto **Hola Mundo**.

Conceptos: widgets

El concepto fundamental en torno al cual gira todo el desarrollo de Flutter es el de [widget](#). De hecho, en Flutter, a excepción de nuestras propias clases para mantener modelos de información y datos, todo será widgets.

[widgets](#)

Son clases Dart cuyos constructores admiten tanto argumentos posicionales como argumentos de nombre, y que nos sirven para representar los elementos de nuestras interfaces.

Los widgets, además, se pueden componer para crear una estructura en forma de árbol que represente la interfaz, y pueden ser widgets sin estado (*stateless*) o con estado (*stateful*), dependiendo de si el widget necesita un estado asociado que pueda estar sujetos a cambios. Por ejemplo, un botón o texto estático, que no cambia y puede declararse constante, sería un buen candidato para ser un widget sin estado, mientras que un componente que va a mostrar el resultado de una llamada asincrónica sería un candidato para ser un widget con un estado.

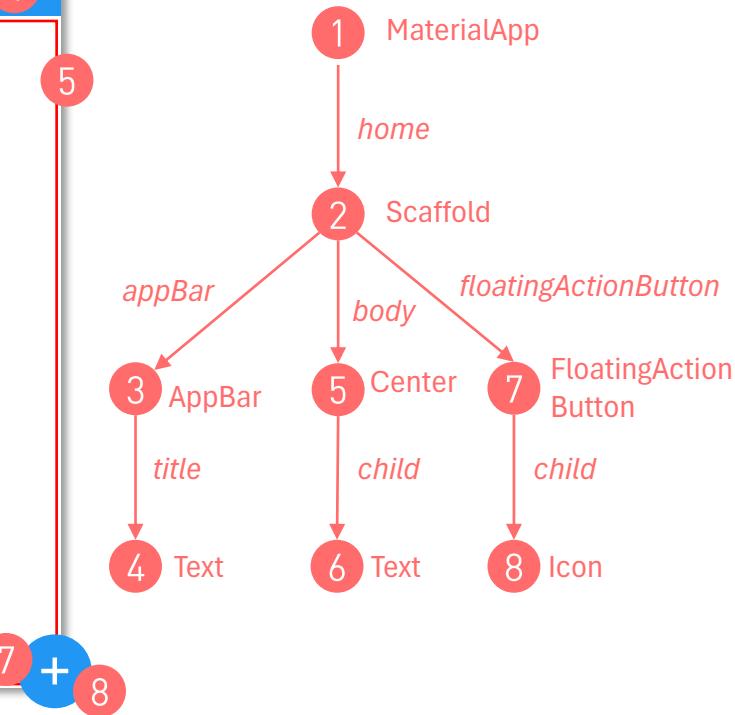
Una diferencia importante entre ambos tipos es que los widgets con estado tienen la capacidad de redibujarse ellos mismos ante cualquier cambio de estado, mientras que los widgets sin estado no pueden hacerlo.

Existe un tercer tipo, los widgets heredados (*InheritedWidget*), que permiten compartir datos entre los componentes que contiene.

Conceptos: árbol de widgets

Los widgets de una aplicación pueden juntarse, de forma que se genere una relación jerárquica en forma de árbol. Podemos ver una aplicación en Flutter como un gran árbol que contiene todos sus widgets. Si en algún momento se produce un cambio en el estado de alguno de los widgets, el propio Flutter es quien se encarga de regenerar el subárbol afectado y redibujar la interfaz.

```
1  MaterialApp(  
  title:'App Mate 1'  
  home: Scaffold(  
    appBar: AppBar(  
      title: const Text(  
        'Barra de aplicaciones'  
      ), // Text  
    ), // AppBar  
    body: const Center(  
      child: Text('Hola Mundo'),  
    ) // Center  
    floatingActionButton: FloatingActionButton(  
      onPressed: () {},  
      child: const Icon(Icons.add),  
    ), // FloatingActionButton  
  ), // Scaffold  
}; // MaterialApp
```



Proyectos

Proyectos

Para crear un nuevo proyecto con Flutter se utiliza el comando `flutter create`, con algunas opciones. Tanto VSCode como Android Studio permiten la creación de proyectos desde su interfaz.

Estructura

Un proyecto típico en Flutter se organiza en diferentes directorios y archivos de configuración y código fuente. Los más importantes son:

El directorio **lib**: Con el código fuente Dart de nuestra aplicación Flutter, que será compilado a código específico de los distintos sistemas. La clase principal será `main.dart` y será el punto de entrada en nuestra aplicación.

El archivo **pubspec.yaml**: Es el archivo de configuración del proyecto Flutter en formato YAML , donde se incluyen las dependencias en bibliotecas, se especifican recursos de imágenes, fuentes, audio o vídeo.

El directorio **android**: Dentro encontraremos la estructura habitual de una aplicación Android, formada por un módulo app, los scripts de Gradle (`build.gradle`, `settings.gradle`), los archivos de Manifiestos (`app/src/main/AndroidManifest.xml`), etc.

El directorio **ios**: Con la estructura necesaria para crear aplicaciones nativas de iOS. En este caso, si deseamos abrir la aplicación para su compilación, modificación o depuración necesitaríamos XCode.

El directorio **web**: Con todo lo necesario para nuestro proyecto para la web.

Además, podemos tener también varios directorios con el proyecto nativo para el resto de plataformas para las que estamos creando la aplicación (`linux` , `windows` , etc.).

Proyectos

Estructura (cont.)

El archivo **.gitignore**: con una configuración predeterminada para omitir ciertos tipos de archivos del control de versiones git.

El archivo **.metadata**: un archivo oculto con las propiedades del proyecto, para su seguimiento.

El archivo **analysis_options.yaml**: on la configuración de análisis estático del código Dart, formado por un conjunto de reglas que servirán para mostrar u ocultar determinados avisos.

El archivo **Read.me**: Se trata de un archivo en formato Markdown con la descripción del proyecto.

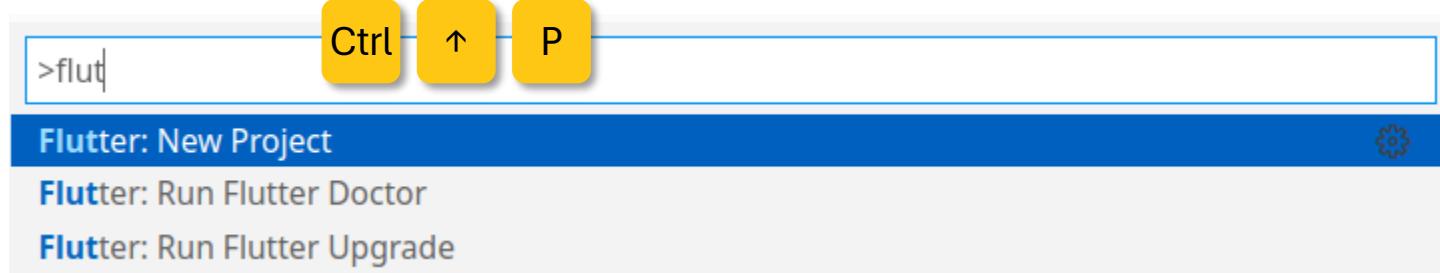
El archivo **pubspec.lock**: mantenido y utilizado por el framework, con información sobre cómo se han generado las dependencias.

El archivo ***iml**: Generalmente con el nombre de la aplicación y extensión iml, no es un archivo propio de Flutter, y junto con la carpeta **.idea**, son archivos propios del IDE IntelliJ.

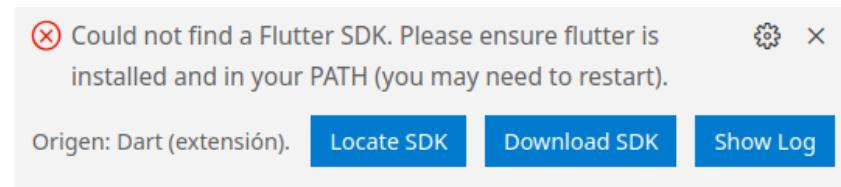
El directorio **tests**: Con las pruebas que deseamos añadir a la aplicación, y con la misma estructura de directorios que utilizamos en **lib**.

Proyectos con VSCode

① Creando el proyecto

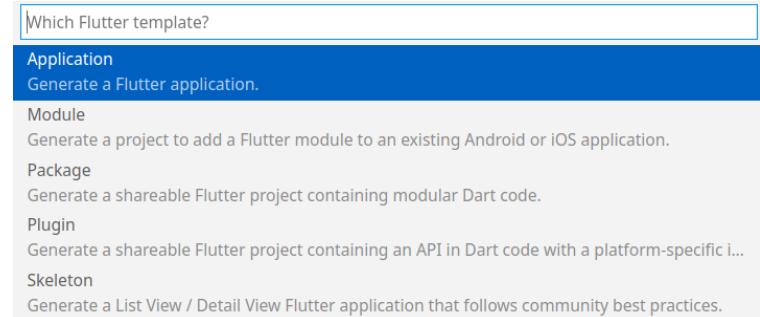


Si VSCode no encontrara directamente el SDK de Flutter, nos mostrará el siguiente mensaje, indicando que le proporcionamos el path a éste:



Desde aquí podremos indicar la carpeta donde se ubica el SDK o descargarlo directamente. También deberá indicarse la misma carpeta para el plugin de Dart

Cuando el SDK está configurado, el asistente nos pide el tipo de aplicación de Flutter y posteriormente el directorio donde guardarla y el nombre del proyecto.



Todo lo anterior generará el orden:

```
flutter create --template app --ios-language swift --android-language Kotlin .
```

Proyectos con VSCode

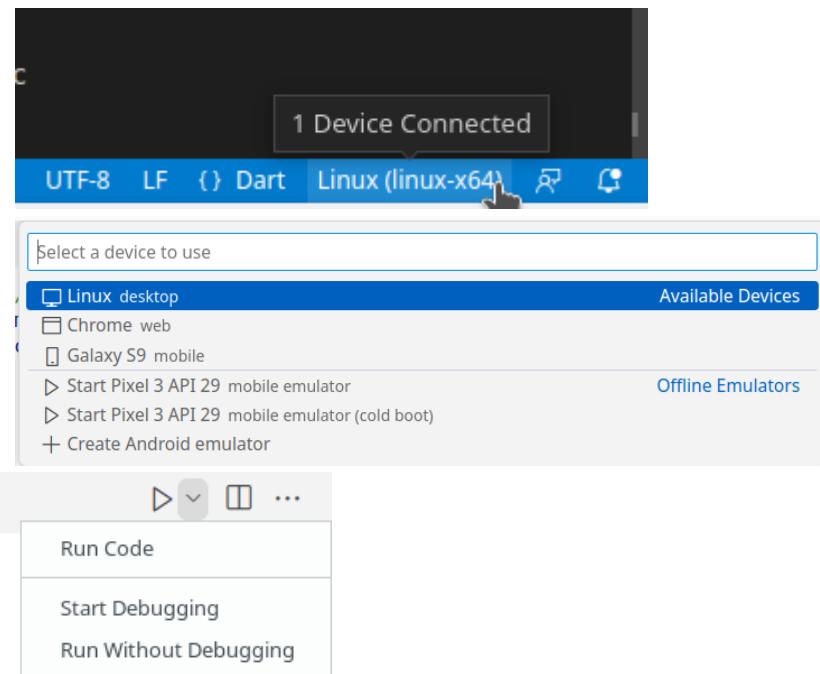
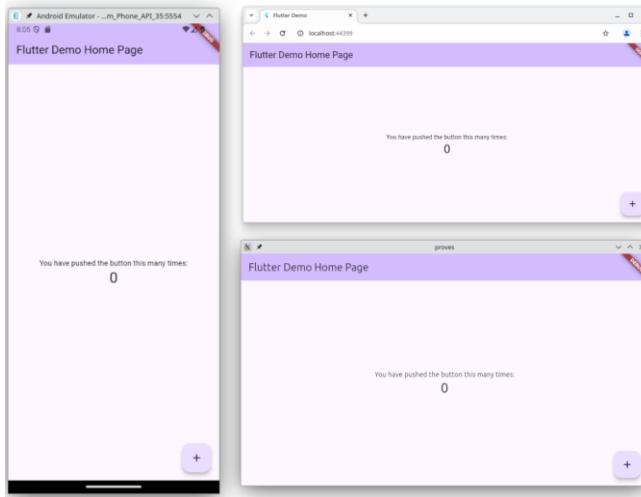
② Comprobación de los dispositivos

En la barra de estado de Visual Studio Code, situada en la parte inferior de la ventana, veremos que se nos muestra el dispositivo configurado de forma predeterminada, en este caso, Linux:

Si hacemos clic sobre él, se nos abrirá en la parte superior un menú para elegir el dispositivo sobre el que queremos ejecutar la aplicación.

③ Lanzando la aplicación

El editor VSCode, en la esquina superior derecha nos muestra los controles para ejecutar o depurar la aplicación.



La primera opción ejecuta el programa que tengamos abierto en el editor, por lo que sólo nos serviría para programas sencillos realizados con Dart. Si lo que queremos es ejecutar el proyecto como tal, debemos hacer uso de las opciones **Start Debugging** (o F5) para iniciar la aplicación en modo depuración, o **Run Without Debugging**, si no queremos ejecutar el depurador.

Proyectos desde el terminal

① Creando el proyecto



Terminal integrada de VSCode

El nombre del proyecto debe ser un nombre de Dart válido, lo que implica que sólo puede estar formado por letras minúsculas, números (excepto el primer carácter), y el guion bajo (_).

```
flutter create <directorio>
```

← Forma fácil (punto . para el directorio actual)

```
flutter create --platforms android,web,linux exemplo_1
```

← Para plataformas específicas

```
$ flutter create --platforms ios exemplo_1/
```

```
Recreating project exemplo_1...
```

← Añadir una nueva plataforma

② Gestión de dispositivos

```
$ flutter devices
Found 2 connected devices:
  Linux (desktop) • linux    • linux-x64          • Ubuntu 24.04.1 LTS 6.8.0-45-generic
  Chrome (web)     • chrome   • web-javascript   • Google Chrome 129.0.6668.89
```

Proyectos desde el terminal

③ Trabajando con emuladores

```
flutter emulators
```

2 available emulators:

Id	Name	Manufacturer	Platform
Medium_Phone_API_35	• Medium Phone API 35	• Generic	• android
Pixel_8_API_35	• Pixel 8 API 35	• Google	• android

To run an emulator, run '`flutter emulators --launch <emulator id>`'.

```
flutter emulators --launch Medium_Phone_API_35
```

```
~/Android/Sdk/emulator/emulator -avd Medium_Phone_API_35
```

← Desde las herramientas de
Android

④ Ejecución de aplicaciones

```
flutter run
```

Connected devices:

```
Linux (desktop) • linux • linux-x64      • Ubuntu 24.04.1 LTS 6.8.0-45-generic
```

```
Chrome (web)   • chrome • web-javascript • Google Chrome 129.0.6668.100
```

```
[1]: Linux (linux)
```

```
[2]: Chrome (chrome)
```

← Si no hay ningún dispositivo móvil en marcha nos
mostrará una lista para elegir

```
$ flutter run --device-id <id_del_dispositivo>
```

← Lanzar el proyecto desde un
dispositivo concreto

JIT y recarga rápida

Cuando tenemos la aplicación en marcha, en la parte superior derecha nos aparece la etiqueta **Debug**, que indica que el compilador está realizando una compilación de tipo JIT o *Just-In-Time*, y se está ejecutando sobre la máquina virtual de Dart, lo que implica que tenemos el **Hot reload** habilitado.

Si hemos lanzado la ejecución desde la línea de comandos, veremos que nos muestra la siguiente información:

```
Launching lib/main.dart on sdk gphone64 x86 64 in debug mode...
Running Gradle task 'assembleDebug'...                                5,3s
✓ Built build/app/outputs/flutter-apk/app-debug.apk.
Syncing files to device sdk gphone64 x86 64...                          119ms
```

Flutter run key commands.

r Hot reload. 🔥🔥🔥 R Hot restart.

h List all available interactive commands.

d Detach (terminate "flutter run" but leave application running).

c Clear the screen

q Quit (terminate the application on the device).

💪 Running with sound null safety 💪

An Observatory debugger and profiler on sdk gphone64 x86 64 is available at:

<http://127.0.0.1:40067/FyBBSiT7H0k=/>

The Flutter DevTools debugger and profiler on sdk gphone64 x86 64 is available at:

<http://127.0.0.1:9100?uri=http://127.0.0.1:40067/FyBBSiT7H0k=/>

JIT y recarga rápida

Como vemos, esta salida se divide en tres bloques:

Un primer bloque donde nos muestra información sobre la construcción del proyecto en modo depuración,

Un segundo bloque, con información sobre diversas órdenes para interactuar con la ejecución del proyecto. Las que más utilizaremos serán las dos primeras:

r, para realizar un *Hot Reload* o recarga de la aplicación *en caliente*. Nos será de gran utilidad cuando hagamos modificaciones estéticas sobre la pantalla actual que no requieren de carga de información ni de otras vistas,

R, para hacer un *Hot Restart*, o un reinicio *en caliente* de la aplicación, por lo que se aplican los cambios realizados en cualquier parte del código en la aplicación, pero sin tener que desinstalarla y volver a instalarla

Y el tercer bloque, aparte de indicar que se está ejecutando con *sound null safety*, nos ofrece dos enlaces web a nuestro equipo, desde los que Flutter nos mostrará información sobre la máquina virtual e información de depuración de nuestra aplicación.

Hola Mundo

Como se lanza Flutter

1

La función `main()` se ejecuta automáticamente

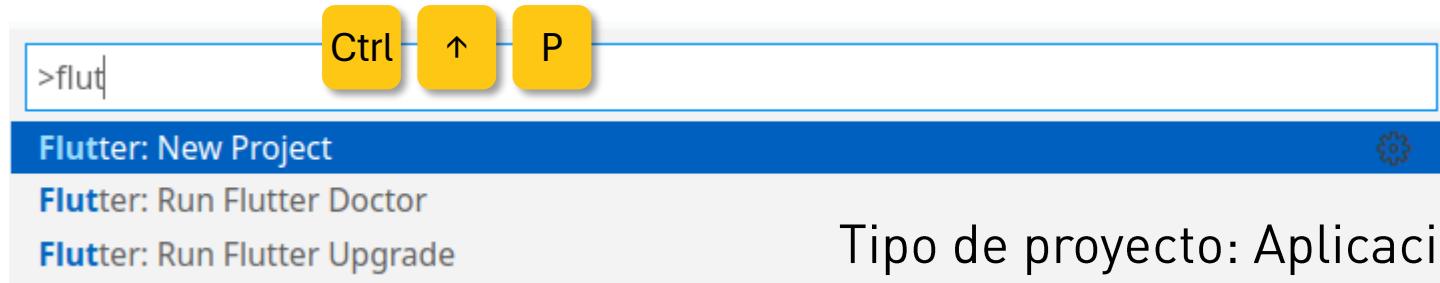
2

Se debe llamar a `runApp()` dentro de la función `main()`

3

El argumento de `runApp()` debe ser el nodo raíz del árbol de widgets

Hola Mundo



Buscar la carpeta donde queramos guardarla y dar un nombre al proyecto

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

void main() {
  runApp(
    const Text(
      'Hola Mundo',
      textDirection: TextDirection.ltr,
    ),
  );
}
```



Hola Mundo

Ctrl .

Run | Debug | Profile

```
void main() {
  runApp(
    const Text(
      'Hola Mundo',
      textDirection: TextDirection.ltr,
    ),
  );
}
```

Extraer...

- Extract Method
- Extract Local Variable
- Extract Widget

Más Acciones...

- Wrap with widget...
- Wrap with Builder
- Wrap with Center
- Wrap with Column
- Wrap with Container
- Wrap with Padding
- Wrap with Row
- Wrap with SizedBox
- Wrap with StreamBuilder

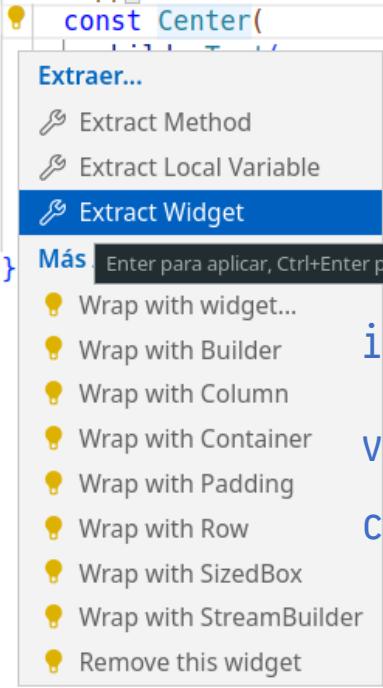
```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const Center(
      const Text(
        'Hola Mundo',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Hola Mundo



```
void main() {  
    runApp(  
        const Center(  
            child: Text('Hola Mundo', textDirection: TextDirection.ltr,  
        ),  
    );  
}
```



```
import 'package:flutter/material.dart';  
  
void main() => runApp(MiApp());  
  
class MiApp extends StatelessWidget {  
    const MiApp({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return const Center(  
            child: Text('Hola Mundo', textDirection: TextDirection.ltr  
        );  
    }  
}
```

Si un widget es CONST todos sus descendientes lo serán también

BuildContext

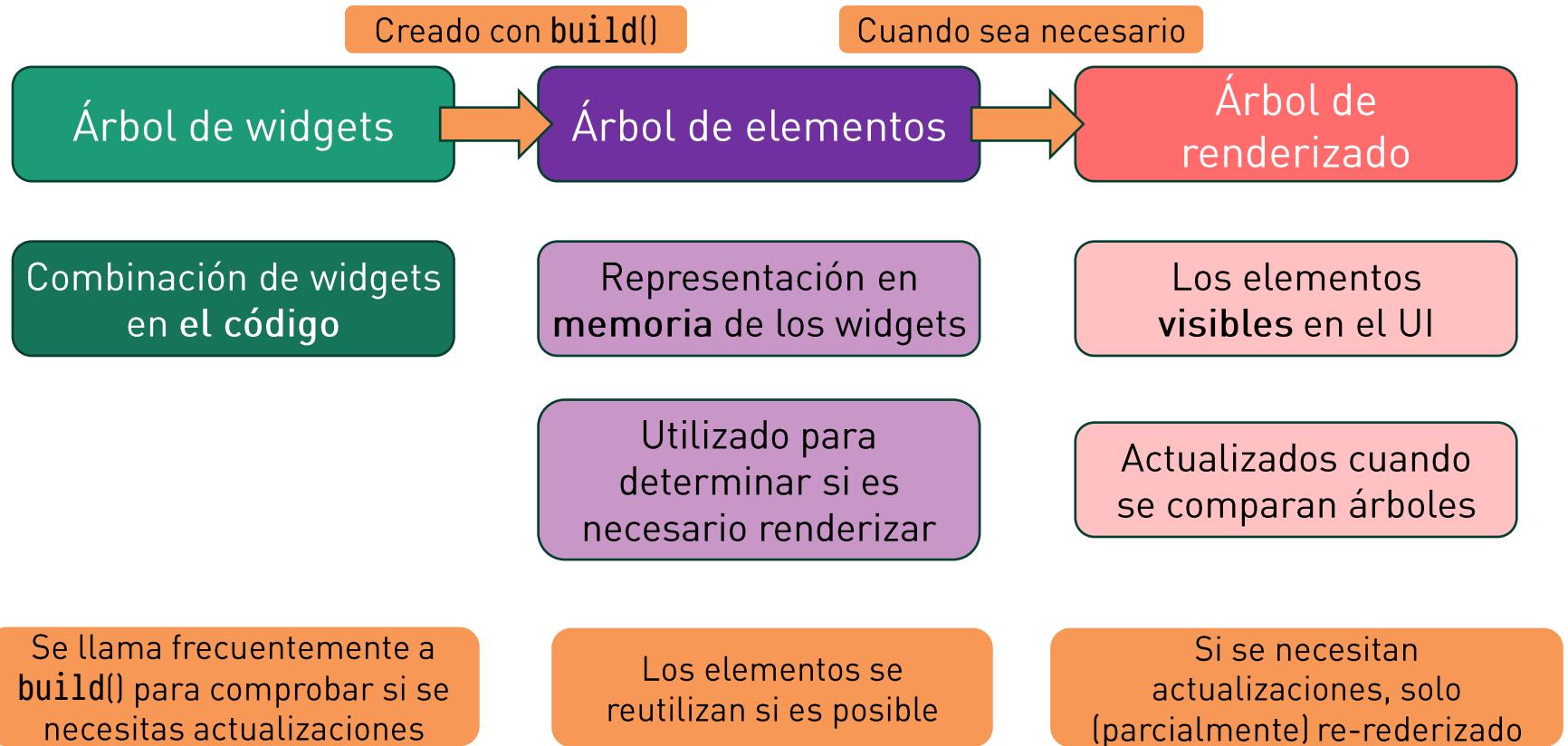
`BuildContext` es un localizador que se utiliza para rastrear cada widget en un árbol y ubicarlo en el mismo. El identificador `BuildContext` de cada widget se pasa a su método `build` que devuelve el árbol de widgets que renderiza un widget.

Cada `BuildContext` es único para un widget. Esto significa que el `BuildContext` de un widget no es el mismo `BuildContext` que el de los widgets que devuelve.

```
class MiApp extends StatelessWidget {  
    const MiApp({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return const Center(  
            child: Text('Hola Mundo', textDirection: TextDirection.ltr  
        )  
    );  
}
```

MiApp tiene su `BuildContext`, Center tiene su `BuildContext` y Text tiene su `BuildContext` todos diferentes entre sí

Key



Key

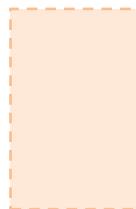
La mayoría del tiempo... no necesitas **keys**. Generalmente, no hay nada de malo en agregarlas, pero también es innecesario. Pero, si te encuentras agregando, removiendo, o reordenando una colección de widgets del mismo tipo que mantiene algún estado, usar **keys** es lo recomendable.

El parámetro **key** se encuentra en cada constructor de widget. La propiedad **key** de un widget conserva el estado cuando los widgets se mueven en el árbol de widgets. En la práctica, esto significa que pueden ser útiles para conservar la ubicación de desplazamiento del usuario o mantener el estado cuando se modifica una colección.

Widgets (intro)

Configuración: app y page

MaterialApp /
CupertinoApp



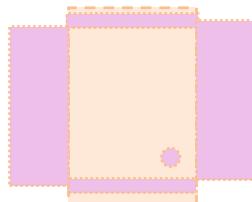
Normalmente el widget raíz de la aplicación

Hace mucho trabajo de configuración “entre bastidores” para la aplicación.

Permite configurar el tema global de la aplicación.

Configura el comportamiento de navegación (pe, animaciones) de la aplicación

Scaffold /
CupertinoPageScaffold



Normalmente se usa como marco para una página de la aplicación.

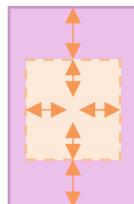
Proporciona un fondo, una barra de la aplicación, pestañas de navegación, etc.

¡Usa solo un scaffold por página!

Disposición

¡Un widget extremadamente versátil!

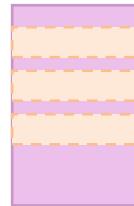
Container



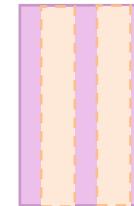
Se puede ajustar el tamaño (ancho, alto, ancho máximo, alto máximo), el estilo (borde, color, forma, etc.) ...

Puede aceptar un widget hijo (aunque no es obligatorio), que también se puede alinear de diferentes maneras.

Row



Column



Imprescindibles si necesitas que varios widgets se ubiquen uno junto al otro horizontal o verticalmente.

Opciones de estilo limitadas → envolver con un contenedor (o envolver widgets hijos) para aplicar estilo.

Los hijos se pueden alinear a lo largo del eje principal y transversal.

Disposición

Container

Solo cuentan con un **child**

Opciones múltiples de alineación y estilos

Ancho flexible (p.e. ancho del child, ancho disponible, etc)

Perfecto para alineaciones y estilos a mediada

Column/Row

Cuentan con múltiples **child** (ilimitados)

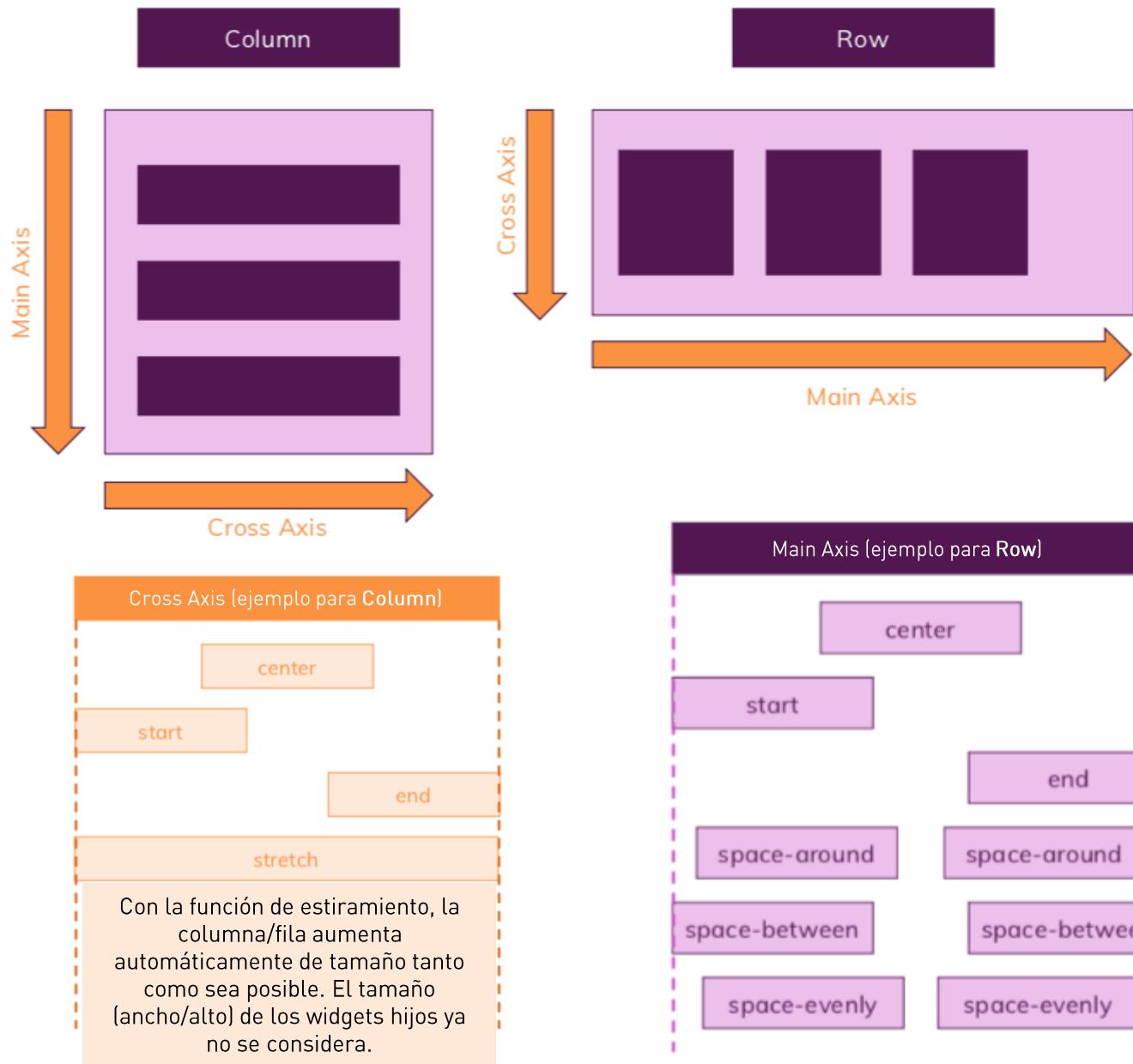
No cuentan con estilos (sí alineación)

Siempre toman todo el espacio disponible: alto en Column y ancho en Row

Se deben usar si los widgets se colocan uno al lado de otro

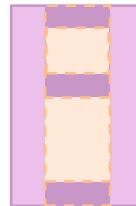


Disposición

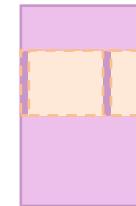


Disposición: hijos de Column/Row

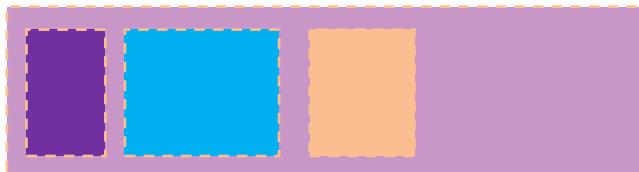
Flexible



Expanded

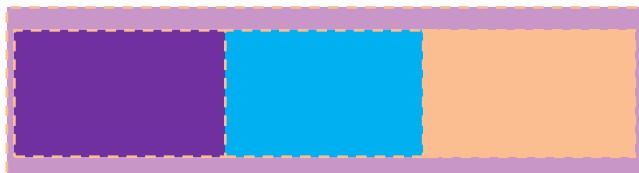


Row() con Flexible (fit:FlexFit.loose) en todos sus hijos



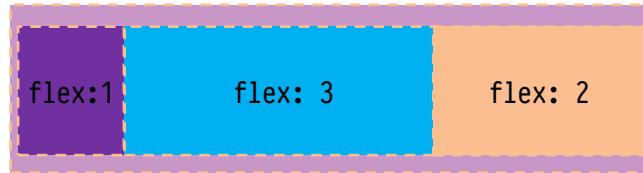
Los widgets son tan grandes como deben ser internamente (por su contenido o tamaño)

Row() con Flexible (fit:FlexFit.tight o Expanded()) en todos sus hijos



Los widgets incrementan su tamaño hasta ocupar todo el padre

Row() con Flexible (fit:FlexFit.tight o Expanded()) en todos sus hijos



Los widgets incrementan su tamaño basado en el valor de la propiedad flex

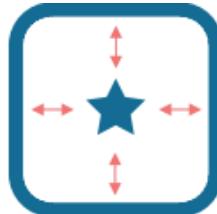
Otros widgets: distribución

Wrap



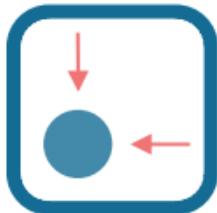
Crea una nueva fila/columna de widgets si el último hijo no cabe en la actual.

Padding



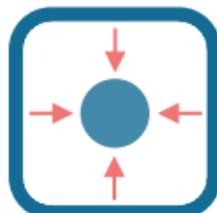
Inserta su widget hijo según los valores de relleno especificados.

Align



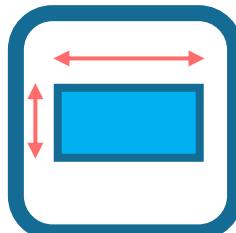
Posiciona su widget hijo y, opcionalmente, se dimensiona según el tamaño del hijo.

Center



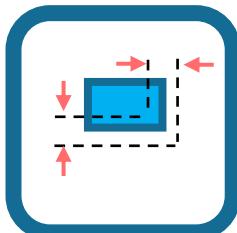
Un widget Align que centra su hijo y, opcionalmente, ajusta su tamaño según el tamaño del hijo.

SizedBox



Una caja con un tamaño específico, que también fuerza a su hijo a que respete su tamaño impuesto.

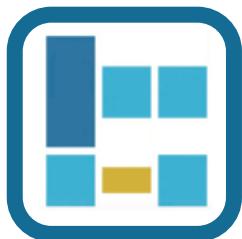
ConstrainedBox



Impone reglas adicionales de tamaño: no ser más grande o más pequeño de ciertas dimensiones, o que tome cierto espacio.

Otros widgets: distribución

Table



Permite ordenar sus hijos en un formato de tabla, filas y columnas de forma estricta.

Transform



Aplica una transformación antes de pintar a su hijo, como escala, traslación y rotación.

if width > 500?

else

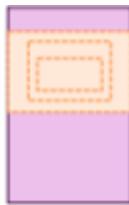


LayoutBuilder

Permite decidir qué árbol de widgets construir en función del tamaño y las restricciones del widget padre.

Contenido

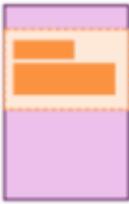
Stack



Se utiliza para colocar elementos uno encima del otro (a lo largo del eje Z) Los widgets pueden superponerse

Puede colocar elementos en el espacio absoluto (es decir, en un espacio de coordenadas) a través del widget `Positioned()`

Card



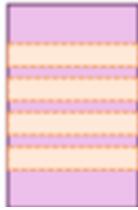
Un contenedor con algún estilo predeterminado (sombra, color de fondo, esquinas redondeadas).

Puede tomar un `child` (puede ser cualquier cosa).

Generalmente se usa para generar una sola pieza/grupo de información.

Elementos repetidos

ListView

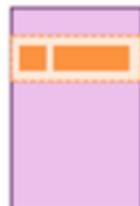


GridView



Se utilizan para generar listas (o cuadrículas) de elementos
Como una Column() pero desplazable (la Columna no lo es).
Se puede disponer verticalmente (predeterminado) y horizontalmente.
Utiliza ListView.builder() para obtener una representación de elementos optimizada para listas muy largas.

ListTile

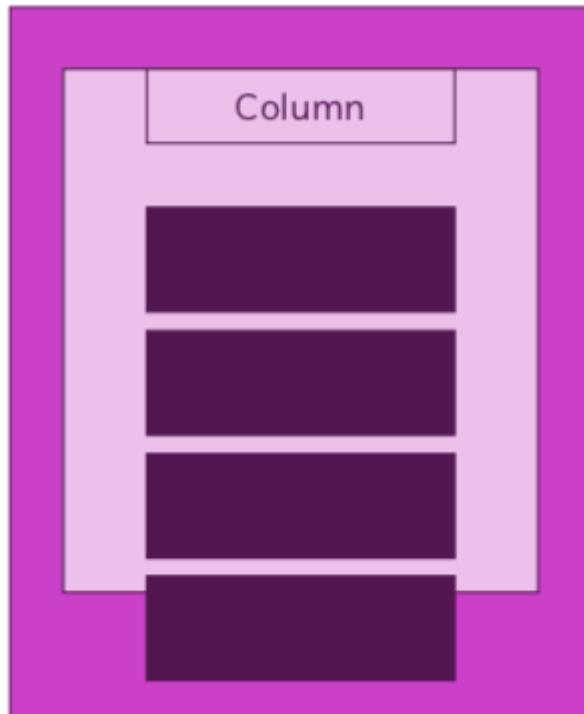


Un Container()/Row() prediseñado que permite lograr una «apariencia de elemento de lista» típica

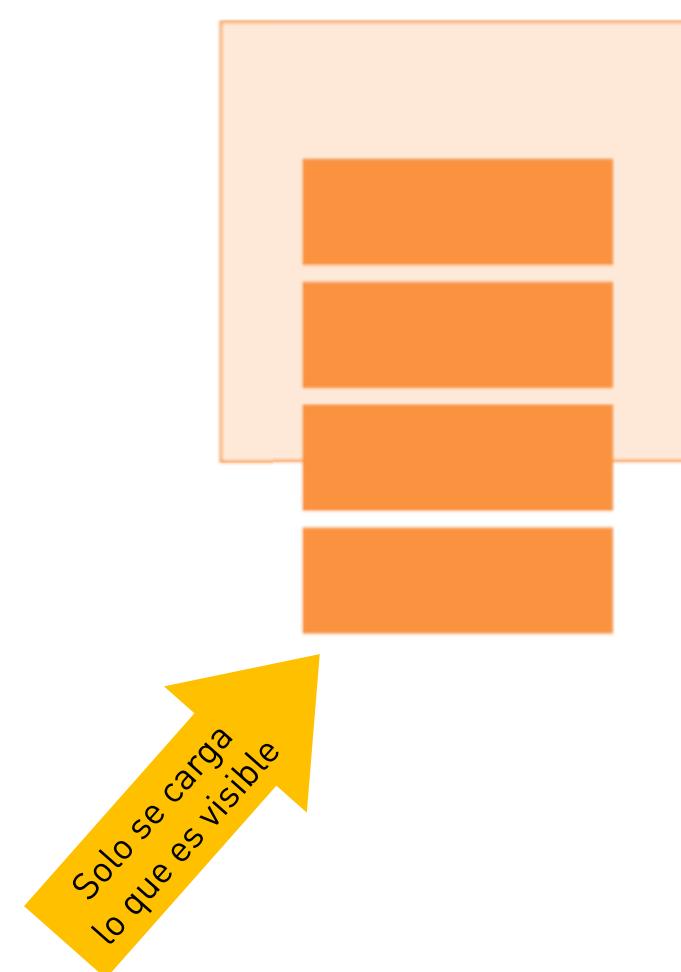
Ofrece varios espacios para widgets (por ejemplo, al principio, un título, al final)

Elementos repetidos

ListView(children[])



ListView.builder()

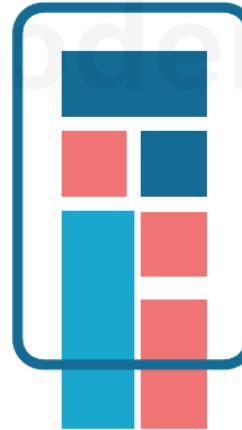


Otros widgets: elementos repetidos



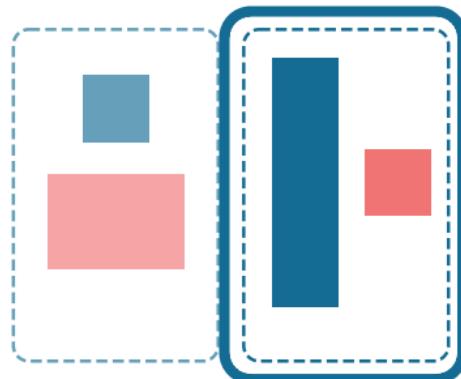
SingleChildScrollView

Un widget que permite desplazar su widget hijo cuando su tamaño supera la ventana gráfica.



CustomScrollView

Un widget que te permite usar **Slivers** para crear efectos de desplazamiento personalizados.

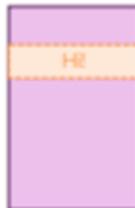


PageView

Un widget desplazable que permite desplazarte página por página, donde cada página es un widget.

Tipos de contenido

Text

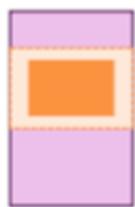


Un widget que muestra texto en la pantalla.

Se puede cambiar el estilo del texto (familia de fuentes, grosor de fuente, tamaño de fuente, etc.).

Se puede controlar el comportamiento del texto (por ejemplo, recortarlo si es demasiado largo).

Image

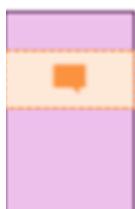


Se utiliza para representar una imagen en la pantalla.

Admite diferentes fuentes (incluidas en la aplicación, imagen web, etc.).

Se puede configurar para cambiar su tamaño de diferentes maneras en un contenedor envolvente.

Icon



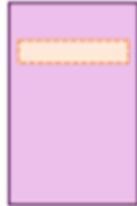
Representa un ícono en la pantalla

Flutter viene con muchos íconos predeterminados de Material (Android) e iOS que puedes usar.

También hay un widget IconButton() en caso de que se necesite un botón con un ícono.

Entrada de usuario

TextField



Representa un campo de texto editable donde el usuario puede ingresar (escribir) información

Muchas, muchas opciones de configuración (autocorrección, mensajes de error, etiquetas, estilos)

Admite diferentes tipos de teclado (correo electrónico, número, texto normal, ...)

ElevatedButton
/TextButton



Botones con diferentes estilos que manejan los toques del usuario

Se puede (y debe) proporcionar una función personalizada que se debe ejecutar al tocar

Se puede diseñar/personalizar

GestureDetector

InkWell



GestureDetector le permite envolver cualquier widget con oyentes táctiles (por ejemplo, doble toque, toque largo)

InkWell hace lo mismo, pero agrega un efecto de onda visual al tocar (el efecto se puede configurar)

Puedes crear sus propios botones/widgets táctiles con estos widgets

Otros widgets: interacción

IconButton



Un widget botón que utiliza un ícono como dato.

Switch



Se utiliza para alternar el estado activado o desactivado de una configuración

Checkbox



Un widget que se utiliza para activar o desactivar un valor de una configuración.

OutlinedButton



Un widget button con un borde de contorno

Slider



Un widget que se utiliza para seleccionar entre un rango de valores continuos o discretos.



KEEP
CALM
AND
ASK
QUESTIONS