

Temas

Estructura de un programa hecho con Flutter

Esquema de la distribución de los componentes (Layout)

Crear programas simples utilizando los componentes básicos del marco de desarrollo Flutter

Crear una aplicación que permite agregar elementos a una lista.



<https://flutter.dev/docs/development/ui/layout>

Coding with Andrea

<https://www.youtube.com/watch?v=RlEnTRBxaSg>

Flutter Layouts

<https://www.javatpoint.com/flutter-layouts>

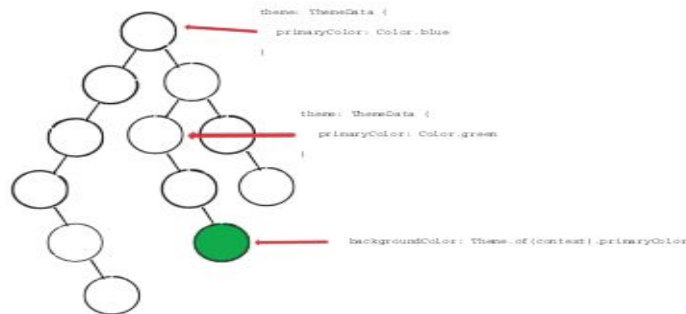
<https://github.com/bizz84/layout-demo-flutter>



BuildContext

BuildContext es otro concepto de Flutter que es crucial para la construcción de aplicaciones, y tiene todo que ver con el seguimiento de todo el árbol de widgets, específicamente, **dónde se encuentran los widgets en el árbol**. Cuando actualizas el tema en tu ThemeData, como hicimos para cambiar el color de la aplicación del contador, se actualizan los widgets hijos en todo el árbol de widgets. **¿Cómo funciona esto?** Está ligado a la idea de BuildContext. Cada método de construcción de un widget toma un argumento, BuildContext, que es una referencia a la ubicación de un widget en el árbol de widgets. Recuerda, la construcción es llamada por el propio marco, así que no tienes que manejar el contexto de construcción por ti mismo, pero querrás interactuar con él a menudo. Un ejemplo concreto es el Theme.of method, un método estático en la clase Theme. Cuando se llama, Theme.of toma un BuildContext como argumento y devuelve la información sobre el tema en ese lugar del árbol de widgets. Por eso, en la aplicación de contador, podemos llamar Theme.of(buildContext).primaryColor para colorear los widgets. Eso obtiene la información del Tema para este punto en el árbol y luego devuelve los datos guardados en la variable primaryColor en la clase Theme. **Cada widget tiene su propio contexto de construcción, lo que significa que si tuvieras varios temas dispersos en el árbol, obtener el tema de un widget podría devolver resultados diferentes a los de otro.** En el caso específico del tema en la aplicación de contador, u otro de los métodos, obtendrás el padre más cercano en el árbol de ese tipo (en este caso, Theme; ver figura 3.7).

El contexto de construcción se utiliza de varias maneras para decirle a Flutter exactamente dónde y cómo ren-der ciertos widgets. Por ejemplo, Flutter utiliza `BuildContext` para mostrar los modales y las rutas. Si se quiere mostrar un nuevo modal, Flutter necesita saber en qué lugar del árbol debe insertarse ese modal. Esto se logra pasando en `BuildContext` a un método que crea modales. Veremos esto en profundidad en el capítulo de rutas. **El punto importante sobre el contexto de construcción, por ahora, es que contiene información sobre el lugar de un widget en el árbol de widgets, no sobre el propio widget.** Se puede decir que los widgets, el estado y el contexto son las tres piedras angulares de la base para desarrollar una aplicación básica en Flutter.



Layout (Diseño de Pantalla)

¿Qué es el Layout?

¿Qué Widgets usamos para Layout?

Layout y Árbol de Widgets

Widgets Column y Row



AppBar

AppBar El widget **AppBar** suele contener el título estándar, la barra de herramientas, las propiedades de las acciones (junto con los botones), así como muchas opciones de personalización.

Title: La propiedad de title se implementa normalmente con un widget de Texto. Puedes personalizarla con otros widgets como el widget DropdownButton. title leading

Leading: La propiedad leading se muestra antes de la propiedad title. Normalmente se trata de un IconButton o BackButton .

Actions: La propiedad actions se muestra a la derecha de la propiedad del título. Es una lista de widgets alineados en la parte superior derecha de un widget de AppBar, normalmente con un IconButton o PopupMenuButton .

flexibleSpace: La propiedad flexibleSpace se apila detrás del widget Toolbar o TabBar. La altura suele ser la misma que la del widget AppBar. Normalmente se aplica una imagen de fondo a la propiedad flexibleSpace, pero se puede utilizar cualquier widget, como un Icono.



Ejemplo con AppBar

```
void main() {  
  //debugPaintSizeEnabled = true; //  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      title: 'Material App',  
      home: Scaffold(  
        appBar: buildAppBar(),  
        body: myLayoutWidget(),  
      ), // Scaffold  
    ); // MaterialApp  
  }  
}
```



```
AppBar buildAppBar() {  
  return AppBar(  
    title: Text('Home'),  
    leading: IconButton(  
      icon: Icon(Icons.menu),  
      onPressed: () {},  
    ), // IconButton  
    actions: <Widget>[  
      IconButton(  
        icon: Icon(Icons.search),  
        onPressed: () {},  
      ), // IconButton  
      IconButton(  
        icon: Icon(Icons.more_vert),  
        onPressed: () {},  
      ), // IconButton  
    ], // <Widget>[]  
  
    //bottom: const PopupMenuButtonWidget(),  
  ); // AppBar  
}
```

Layout

Es el esquema de distribución de los componentes(Widget), dentro un diseño de una pantalla.

En Flutter, casi todo es un widget -> incluso los modelos de layout son widgets. Las imágenes, iconos, y texto que ves en una app Flutter, son todo widgets.

Pero hay cosas que no se ven que también son widgets, como: las filas, columnas, cuadrículas que organizan, restringen, y alinean los widgets visibles.

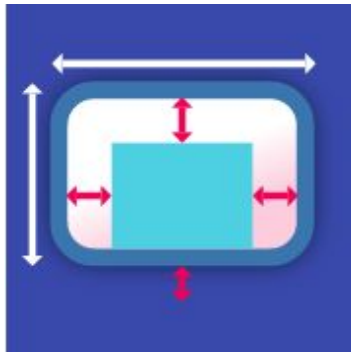
Se crea un layout mediante la composición de widgets para construir widgets más complejos.

Tipos de Layout

Layout->Es el esquema de disposición de los Widgets en pantalla.
Diseño-maquetado

**Widgets de layout
(diseño) para un solo
Widget Hijo**

Container



```
Container(  
  child: Text('Hello'),
```

**Widgets de layout
(diseño) para
múltiples hijos**

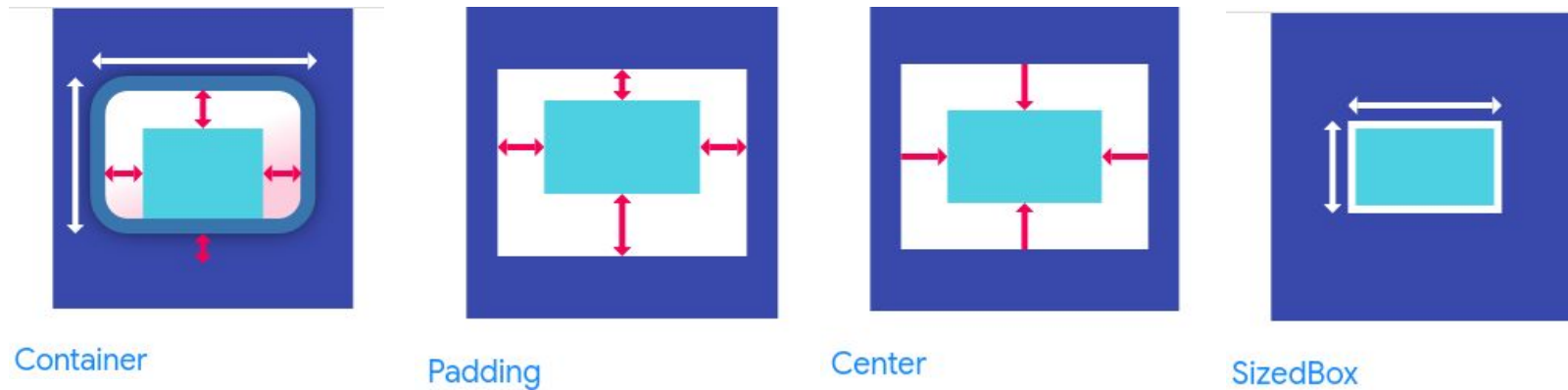
Row



```
Row(  
  children: <Widget>[  
    Text('Hello'),
```

Fuente: <https://flutter.dev/docs/development/ui/widgets/layout>

Widgets de Layout->De un solo hijo



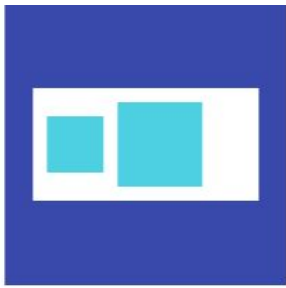
```
Container(  
  child: Text(""),  
)
```



Se le puede asignar un Widget hijo

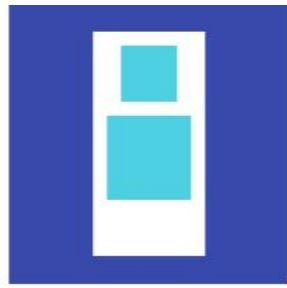
Fuente: <https://esflutter.dev/docs/development/ui/widgets/layout>

Widgets de Layout->De múltiples hijos



Row

Layout con una lista de widgets hijos en dirección horizontal.



Column

Layout con una lista de widgets hijos en dirección vertical.



Expanded


Un widget que expande un hijo de un Row, Column, o Flex.



ListView

Una lista lineal y desplazable de widgets. ListView es el widget de desplazamiento más utilizado. Muestra a sus hijos uno tras otro en la dirección de desplazamiento. En el eje transversal, se requiere que los hijos llenen el ListView.

Row(

children: [
Text("Primer Fila"),
Text("Segunda Fila"),
]

**Se le puede asignar varios
Widgets hijo**

),

Fuente: <https://esflutter.dev/docs/development/ui/widgets/layout>

Scaffold

El widget **Scaffold** implementa el layout(diseño visual) básico de Material Design, permitiéndole añadir fácilmente varios widgets como **AppBar** , **BottomAppBar** , **FloatingActionButton** , **Drawer** , **SnackBar** , **BottomSheet** , y más.

```
void main() {  
  //debugPaintSizeEnabled = true; //  
  runApp(MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      title: 'Material App',  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Ejercicios de layout'),  
        ), // AppBar  
        body: myLayoutWidget(),  
      ), // Scaffold  
    ); // MaterialApp  
  }  
}
```

Píxeles Lógicos en Flutter (logical pixels)

Flutter trabaja a nivel de píxeles.

Los **píxeles lógicos** tienen aproximadamente el mismo tamaño visual en todos los dispositivos.

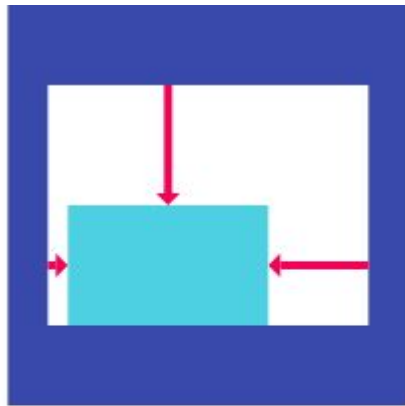
Los **píxeles físicos** son del tamaño de los píxeles reales de hardware del dispositivo.

El número de **píxeles físicos** por cada **píxel lógico** se describe en la relación de píxeles del dispositivo. Se puede ver con MediaQuery

```
var medida = MediaQuery.of(context).devicePixelRatio;
```

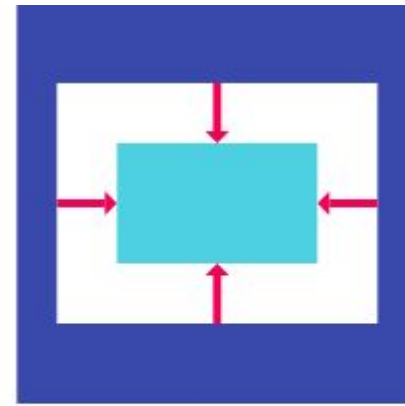
*El número de píxeles del dispositivo para cada píxel lógico. Este número podría no ser una potencia de dos. De hecho, podría no ser ni siquiera un número entero. Por ejemplo, el Nexus 6 tiene una proporción de píxeles de dispositivo de 3,5

Center y Align



Align

Un widget que alinea a su hijo dentro de sí mismo y, opcionalmente, se dimensiona en función del tamaño del child.



Center

Un widget que centra su hijo en sí mismo.



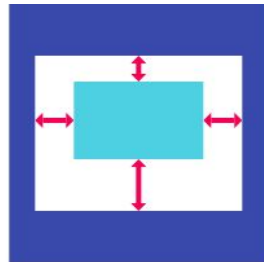
Relleno (Padding) y Margen (Margin)

En Flutter, algunos widgets tienen un relleno y una propiedad de margen para trabajar con el espacio.

- El relleno(Padding) es el espacio entre el contenido y el borde de un widget (que también puede no ser visible).
- El margen(Margin) es el espacio fuera del borde.



Padding



Padding: Relleno
Edge Inset: Borde del Cuadrado

Padding

Usar Padding para agregar relleno a la izquierda, arriba, a la derecha, abajo o todos los lados de su widget hijo.

Solo hay que tomar el widget al que se quiere dar espacio y ponerlo como hijo de **Padding**. Luego utilizar la propiedad **padding** para definir el espacio. Al pasar **EdgeInsets.all()**, por ejemplo, va a poner la misma cantidad de relleno en todos los lados del widget hijo.

```
Padding(  
  padding: EdgeInsets.all(12.0),  
  child: BlueBox(),  
)
```



Para elegir el lado, o lados, para el desplazamiento, puedes usar el constructor **only()**.

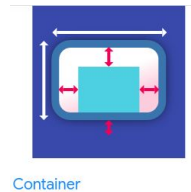
```
EdgeInsets.only(right:80)
```

Permite crear rellenos verticales y horizontales simétricos

```
EdgeInsets.symmetric(vertical:48.5)
```

El constructor **EdgeInsets.all** crea un espacio en los cuatro lados de un cuadro: arriba, derecha, abajo, y a la izquierda. En el **ejemplo**, crea un desplazamiento de **12 píxeles lógicos** en todos los lados de un Container:

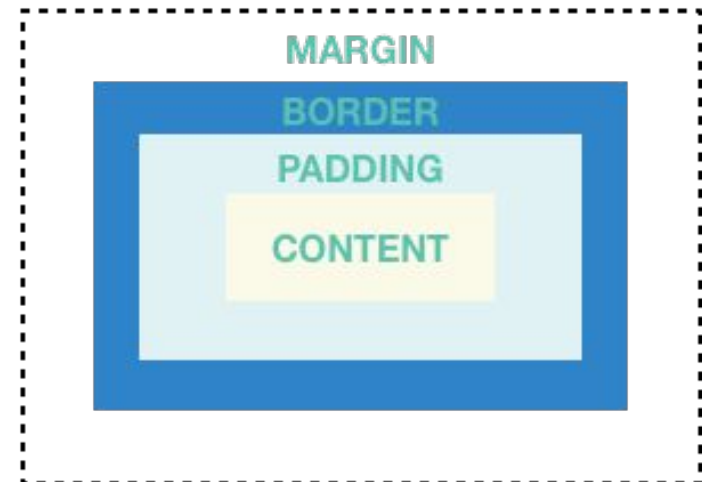
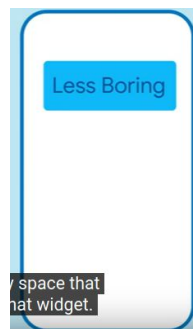
Container



Este widget puede ser usado para personalizar su widget hijo. Usar Container para añadir: alto, ancho, padding, márgenes, bordes, , color, transformar (rotar, mover, sesgar), y muchas más propiedades.

- Añade padding, márgenes, bordes
- Cambia color o imagen de fondo
- Contiene un único hijo, pero este hijo puede ser un Row, Column, o incluso la raíz del árbol de widget

```
Container(  
  child: Text('Less boring'),  
  color: Colors.blue,  
  padding: EdgeInsets.all(20.0),  
  margin: EdgeInsets.all(20.0),  
);
```



Container->Ejemplo

```
Widget myLayoutWidget() {  
  return Container(  
    margin: EdgeInsets.all(30.0),  
    padding: EdgeInsets.all(10.0),  
    alignment: Alignment.topCenter,  
    width: 200,  
    height: 100,  
    decoration: BoxDecoration(  
      color: Colors.green,  
      border: Border.all(),  
    ),  
    child: Text("Hello", style: TextStyle(fontSize: 30)),  
  );  
}
```

SizedBox

Este widget obliga a su hijo a tener una anchura y/o altura específica (asumiendo que los valores son permitidos por el padre de este widget).

```
Column(  
  children: [  
    MyButton(),  
    SizedBox(height: 200),  
    OtherButton(),  
  ],  
)
```

Puedes usar SizedBox sin un hijo. No renderizará nada, pero seguirá ocupando el espacio. Ideal para agregar espacios entre los widgets.

```
SizedBox(  
  width: 200,  
  child: MyButton(),  
)
```

```
SizedBox(  
  width: 200,  
  height: 100,  
  child: MyButton(),  
)
```

Al poner infinito en ambas dimensiones, el hijo llena el espacio disponible.

```
SizedBox(  
  width: double.infinity,  
  height: 100,  
  child: MyButton(),  
)
```

```
SizedBox(  
  width: double.infinity,  
  height: double.infinity,  
  child: MyButton(),  
)
```

```
SizedBox.expand(  
  child: MyButton(),  
)
```

Expanded

Este widget expande y llena el espacio disponible para el widget hijo que pertenece a un widget: Column, Row o Flex.

Se expanden para llenar el espacio disponible a lo largo del eje principal (por ejemplo, horizontalmente para una Fila o verticalmente para una Columna). Si se expanden varios hijos, el espacio disponible se divide entre ellos según el factor de flexibilidad.

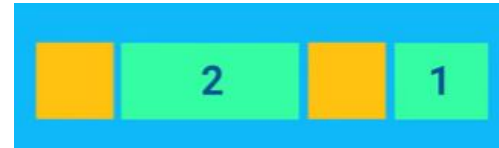
```
Row(  
  children: [  
    MyWidget(),  
    MyWidget(),  
    MyWidget(),  
  ],  
)
```



```
Row(  
  children: [  
    MyWidget(),  
    Expanded(  
      child: MyWidget()  
    ),  
    MyWidget(),  
  ],  
)
```



```
Expanded(  
  flex: 2,  
  child: Container()  
)
```



MediaQuery

Una forma fácil y rápida de obtener las medidas de la pantalla en Flutter

```
double width = MediaQuery.of(context).size.width;  
double height = MediaQuery.of(context).size.height;
```

El tamaño de los dispositivos en píxeles lógicos (por ejemplo, el tamaño de la pantalla).

Los píxeles lógicos tienen aproximadamente el mismo tamaño visual en todos los dispositivos. Los píxeles físicos son del tamaño de los píxeles reales de hardware del dispositivo. El número de píxeles físicos por cada píxel lógico se describe mediante la [relación entre píxeles del dispositivo]:

```
var relacionPixeles= MediaQuery.of(context).devicePixelRatio;
```

MediaQuery.of(context).padding devuelve un Padding que proporciona los detalles del padding para la propia aplicación; es decir, el relleno entre el borde de la pantalla del dispositivo y el widget de nivel superior.

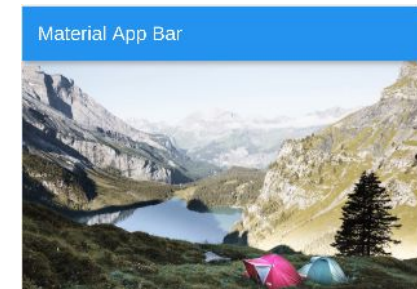
Imágenes

```
{-} pubspec.yaml x  
{-} pubspec.yaml  
43 # included with your application  
44 # the material Icons class.  
45 uses-material-design: true  
46  
47 # To add assets to your application  
48 assets:  
49   - images/lake.jpg
```



```
Image.asset(  
  'images/lake.jpg'  
  width: 600,  
  height: 240,  
  fit: BoxFit.cover,  
) , // Image.asset
```

BoxFit.cover:
Cubre toda la
caja



El widget Image muestra una imagen de una fuente local o URL (web).

Image.asset()

➤ ➤ Image.file()

➤ ➤ Image.memory()

Image.network() —Obtiene Imagen de una URL

Imágenes

```
class Layout1 extends StatelessWidget {  
  const Layout1({Key key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    var medida = MediaQuery.of(context);  
  
    return Column(  
      children: [  
        Container(  
          height: medida.size.height * 0.40,  
          width: medida.size.width - 20,  
          child: Image.asset(  
            'images/pavlova.jpg',  
            fit: BoxFit.cover,  
          ), // Image.asset  
        ), // Container  
        Container(  
          height: medida.size.height * 0.40,  
          width: medida.size.width - 20,  
          child: Image.network(  
            'https://picsum.photos/250?image=9',  
            fit: BoxFit.cover,  
          ), // Image.network  
        ), // Container  
      ],  
    ); // Column  
  }  
}
```

Image.asset(
 'images/pavlova.jpg',
 fit: BoxFit.cover,
),

Image.network(
 'https://picsum.photos/250?image=9',
 fit: BoxFit.cover,
),



Fuente: <https://www.woolha.com/tutorials/flutter-display-and-adjust-images-from-assets>

Row

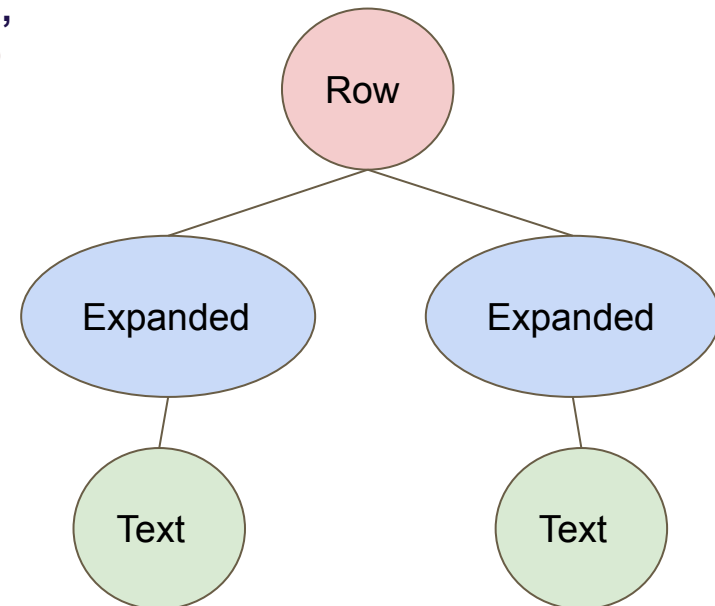
Un widget que muestra a sus hijos en una matriz horizontal.

Para hacer que un hijo se expanda para llenar el espacio horizontal disponible, envuelva al hijo en un widget Expanded.

El widget de Row no se desliza(no scroll) (y en general se considera un error tener más hijos). Si tiene una fila de widgets y quiere que puedan desplazarse, si no hay espacio suficiente, considere la posibilidad de usar un ListView.

Fuente: <https://api.flutter.dev/flutter/widgets/Row-class.html>

```
Row(  
  children: <Widget>[  
    Expanded(  
      child: Text('Text1', textAlign: TextAlign.center),  
    ),  
    Expanded(  
      child: Text('Text2', textAlign: TextAlign.center),  
    ),  
  ],  
)
```



Column

Un widget que muestra a sus hijos en una matriz vertical.

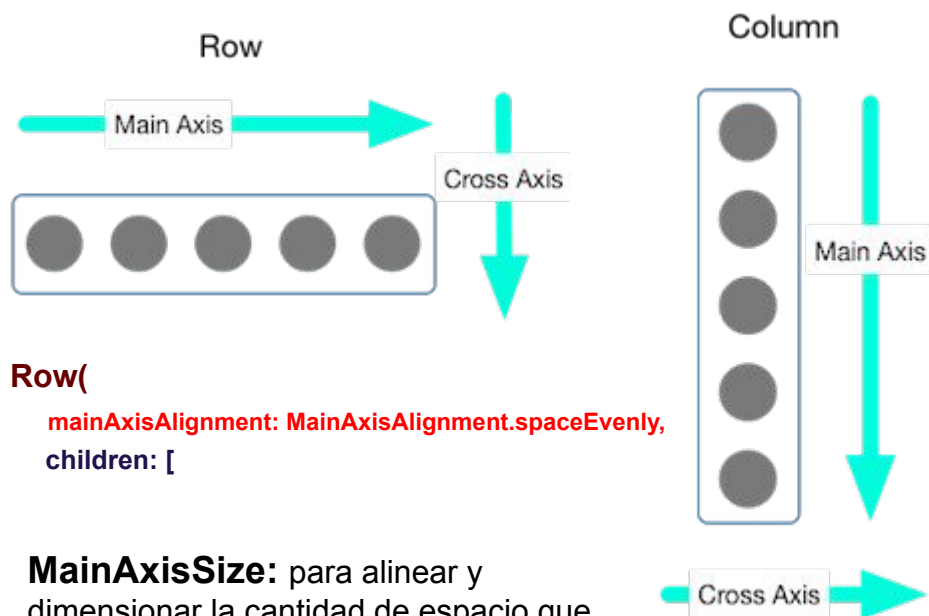
Para hacer que un hijo se expanda o se amplie para llenar el espacio vertical disponible, envuelva al hijo en un widget **Expanded**.

El widget de Column no se desplaza(no scroll) (y en general se considera un error tener más hijos). Si tiene una columna de widgets y quiere que se desplacen(scroll), si no hay espacio suficiente, considere la posibilidad de usar un **ListView**.

```
Column(  
  children: <Widget>[  
    Text('Deliver features faster'),  
    Text('Craft beautiful UIs'),  
    Expanded(  
      child: FittedBox(  
        fit: BoxFit.contain, // de lo contrario  
        el logotipo será diminuto  
        child: const FlutterLogo(),  
      ),  
    ),  
  ],  
)
```

Fuente: <https://api.flutter.dev/flutter/widgets/Column-class.html>

Alineación de Widgets



Row(

`mainAxisAlignment: MainAxisAlignment.spaceEvenly,`
`children: [`

MainAxisSize: para alinear y dimensionar la cantidad de espacio que se ocupa en el eje principal.

La propiedad `mainAxisAlignment` determina cómo Row and Column puede posicionar a sus hijos en ese espacio extra.

MainAxisAlignment.start

Posiciona a los widget/s hijo/s cerca del comienzo del eje principal. (Izquierda para la fila, arriba para la columna)

MainAxisAlignment.end

Posiciona a los widget/s hijo/s cerca del final del eje principal. (Derecha para la fila, abajo para la columna)

MainAxisAlignment.center

Posiciona a los widget/s hijo/s en el centro del eje principal.

MainAxisAlignment.spaceBetween

Divide el espacio extra de manera uniforme entre los widget/s hijo/s

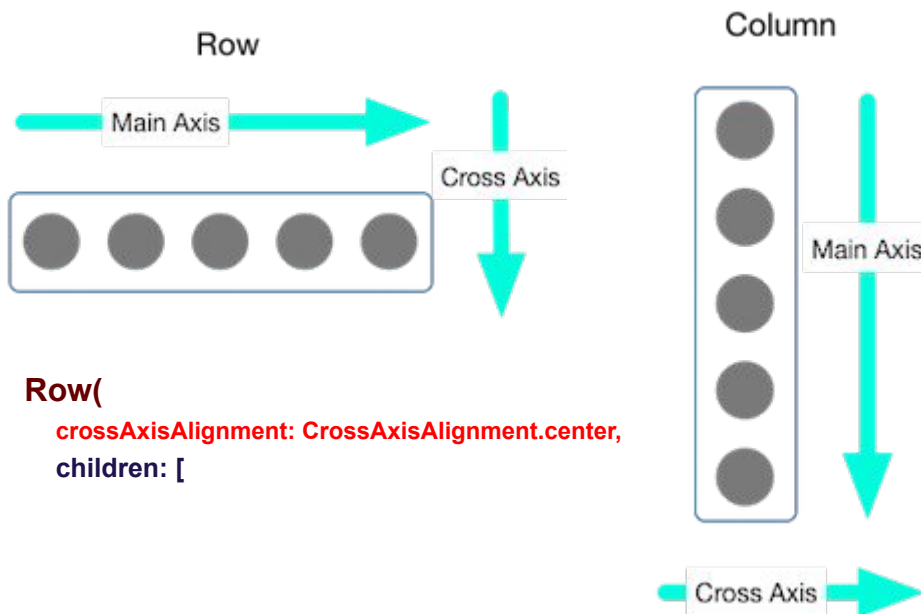
MainAxisAlignment.spaceEvenly

Divide el espacio extra uniformemente entre los widget/s hijo/s y antes y después de los widget/s hijo/s

MainAxisAlignment.spaceAround

Similar a `MainAxisAlignment.spaceEvenly`, pero reduce la mitad del espacio antes del primer widget hijo y después del último widget. Reduce a la mitad del ancho entre los widget/s hijo/s.

Alineación de Widgets



Row(

crossAxisAlignment: `CrossAxisAlignment.center`,
children: [

La propiedad `crossAxisAlignment` determina cómo Row and Column pueden posicionar a sus hijos en su eje transversal.

CrossAxisAlignment.start

Posiciona a los widgets/s hijo/s cerca del comienzo del eje transversal. (Arriba para la fila, a la izquierda para la columna)

CrossAxisAlignment.end

Posiciona a los widgets/s hijo/s cerca del final del eje transversal. (Abajo para la fila, a la derecha para la columna)

CrossAxisAlignment.center

Posiciona a los widgets/s hijo/s en el centro del eje transversal. (Medio para la fila, centro para la columna)

CrossAxisAlignment.stretch

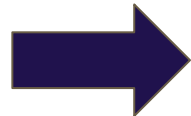
Estira a los widgets/s hijo/s a través del eje transversal. (De arriba a abajo para la fila, de izquierda a derecha para la columna)

CrossAxisAlignment.baseline

Alinea a los widgets/s hijo/s por sus líneas de base de carácter. (Sólo para la clase de texto, y requiere que la propiedad `TextBaseline` se establezca en `TextBaseline.alfabético`).

Dimensionar los widgets

Cuando la disposición de los elementos es demasiado grande para encajar en un dispositivo, aparece un patrón de rayas amarillas y negras a lo largo del borde afectado (overflowing by X pixels). Aquí hay un ejemplo de una fila que es demasiado ancha:



```
Row(  
  children: [  
    Expanded(child: Icon(Icons.home)),  
    Expanded(child: Icon(Icons.home)),  
  ],  
);
```

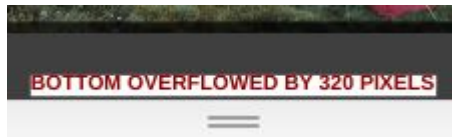


SingleChildScrollView

Añade la capacidad de desplazamiento vertical u horizontal a un widget hijo

Es especialmente útil cuando tu widget puede ocupar más espacio que el espacio disponible en la pantalla y quieres habilitar el desplazamiento para el contenido desbordante.

También es útil si se necesita encoger en ambos ejes (la dirección principal de desplazamiento así como el eje transversal), como se podría ver en un diálogo o menú emergente.



Widgets Row y Column

Row y Column son widgets primitivos básicos para layout horizontales y verticales—estos widgets de bajo nivel permiten una personalización máxima.

*Flutter también ofrece widgets especializados de alto nivel que deben ser suficientes para tus necesidades. Por ejemplo, en lugar de un **Row** podrías preferir **ListTile**, un widget de fácil uso con propiedades para iconos iniciales y finales, y hasta 3 líneas de texto.

*En lugar de **Column**, podrías preferir **ListView**, un layout de tipo columna que permite el scroll automático si su contenido es demasiado largo para ajustarse al espacio disponible.

SafeArea

En los dispositivos actuales, las apps rara vez funcionan como un rectángulo. Pueden aparecer barras, controles de notificación esquinas redondeadas y muescas que cortan el contenido y complican los diseños.

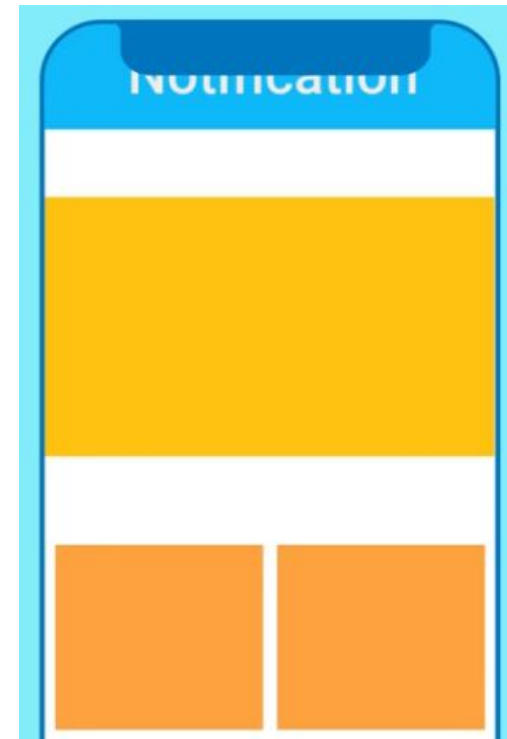
SafeArea usa una MediaQuery para ver las dimensiones de la pantalla y rellenar para que se adapte.

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

```
      SafeArea(
        child: ListView(
          children: List.generate(
            100,
            (i) => Text('This is some text'),
          ),
        ),
      ),
    ),
  );
}
```

```
@override
Widget build(BuildContext context) {
  return SafeArea(
    child: Scaffold(
      appBar: AppBar(
```

Si tu app se ve así, rodéala en una SafeArea, y protege tus contenidos tanto en iOS como en Android.



Tips para Layout

Las Row y Column son Widgets básicos primitivos para diseños horizontales y verticales; estos Widgets de bajo nivel permiten una máxima personalización.

Flutter también ofrece widgets especializados de nivel superior que pueden ser suficientes para sus necesidades. Por ejemplo:

- En lugar de **Row** podrías preferir **ListTile**, un widget fácil de usar con propiedades para los iconos principales y secundarios, y hasta 3 líneas de texto.
 - En lugar de **Column**, puede que prefieras **ListView**, un diseño en forma de columna que se desplaza automáticamente si su contenido es demasiado largo para ajustarse al espacio disponible.
-

ListView.builder

ListView: Este constructor es apropiado para una lista con pocos Widgets.

ListView.builder: Este constructor es apropiado para listas con un gran (o infinito) número de hijos porque el constructor se llama sólo para aquellos hijos que son realmente visibles.

```
final List<String> entries = <String>['A', 'B', 'C'];  
final List<int> colorCodes = <int>[600, 500, 100];
```

```
ListView.builder(  
  padding: const EdgeInsets.all(8),  
  itemCount: entries.length,  
  itemBuilder: (BuildContext context, int index) {  
    return Container(  
      height: 50,  
      color: Colors.amber[colorCodes[index]],  
  
      child: Center(child: Text('Entry ${entries[index]}')),  
    );  
  }  
);
```


Card

Una tarjeta de diseño material: un panel con esquinas ligeramente redondeadas y una sombra de elevación.

Una tarjeta es una hoja de material que se utiliza para representar alguna información relacionada, por ejemplo, un álbum, una ubicación geográfica, una comida, datos de contacto, etc.



Widgets Comunes

SingleChildScrollView —This adds vertical or horizontal scrolling ability to a single child widget.

Padding —This adds left, top, right, and bottom padding.

Column —This displays a vertical list of child widgets.

Row —This displays a horizontal list of child widgets.

Container —This widget can be used as an empty placeholder (invisible) or can specify height, width, color, transform (rotate, move, skew), and many more properties.

Expanded —This expands and fills the available space for the child widget that belongs to a Column or Row widget.

Widgets Comunes

Text —The Text widget is a great way to display labels on the screen. It can be configured to be a single line or multiple lines. An optional style argument can be applied to change the color, font, size, and many other properties. Building the Full Widget Tree

Stack —What a powerful widget! Stack lets you stack widgets on top of each other and use a Positioned (optional) widget to align each child of the Stack for the layout needed. A great example is a shopping cart icon with a small red circle on the upper right to show the number of items to purchase.

SafeArea El widget SafeArea es necesario para los dispositivos actuales como el iPhone X o los dispositivos Android con una muesca (un recorte parcial que oscurece la pantalla que suele estar en la parte superior del dispositivo). El widget de SafeArea añade automáticamente suficiente relleno al widget infantil para evitar las intromisiones del sistema operativo. Opcionalmente, puede pasar una cantidad mínima de relleno o un valor booleano para no forzar el relleno en la parte superior, inferior, izquierda o derecha.

Widgets Comunes

Buttons: Hay una variedad de botones para elegir en diferentes situaciones como RaisedButton , FloatingActionButton , FlatButton , IconButton , PopupMenuButton , and ButtonBar .

Capitulo 10: SingleChildScrollView , SafeArea , Padding , Column , Row , Image , Divider , Text , Icon , SizedBox , Wrap , Chip , and CircleAvatar .



Discusión de Diseño

¿Por qué usar un widget de Relleno en lugar de un Contenedor con una propiedad de Relleno de Contenedor?

No hay realmente ninguna diferencia entre los dos. Si proporciona un argumento de `Container.padding`, `Container` simplemente construye un widget de `Padding` para usted.

`Container` no implementa sus propiedades directamente. En su lugar, `Container` combina varios widgets más simples en un cómodo paquete. Por ejemplo, la propiedad `Container.padding` hace que el contenedor construya un widget de `Padding` y la propiedad `Container.decoration` hace que el contenedor construya un widget `DecoratedBox`. Si considera que `Container` es conveniente, no dude en utilizarlo. Si no, no dude en construir estos widgets más sencillos en cualquier combinación que satisfaga sus necesidades.

De hecho, la mayoría de los widgets de Flutter son simplemente combinaciones de otros widgets más simples. La composición, más que la herencia, es el principal mecanismo para construir widgets.

Por lo tanto, no hay que preocuparse por el relleno de los widgets. Una buena práctica para conseguir algunos espacios dentro de una `Column()` o `Fila()` es usar `SizedBox()`. Entonces añades un espacio extra que se va acumulando a lo largo o a lo ancho.

Formas de Mejorar el Código->(Refactorizar)

Para crear un layout se van anidando los widgets para crear una interfaz de usuario personalizada. El resultado de agregar los widgets es llamado el árbol de los widgets. A medida que el número de widgets aumenta, el árbol de widgets comienza a expandirse rápidamente y hace que el código sea difícil de leer y manejar.

Usando const

Con un Método

Clase Widget



Refactorizar con Clase Widget

Refactorizar con una clase de widget permite crear el widget como subclase de StatelessWidget.

Se pueden crear widgets reutilizables dentro del archivo actual o separado de Dart e iniciarlos en cualquier lugar de la aplicación.

El constructor debe comenzar con una palabra clave `const`, lo que te permite guardar en caché y reutilizar el widget. Cuando llames al constructor para que inicie el widget, usa la palabra clave `const`. Al llamar con la palabra clave `const`, el widget no se reconstruye cuando otros widgets cambian de estado en el árbol. Si omites la palabra clave `const`, el widget será llamado cada vez que el widget padre se redibuje.

La clase de widget se basa en su propio `BuildContext`, no en el padre como el uso de una constante y o el método

`.BuildContext` es responsable de manejar la ubicación de un widget en el árbol de widgets.

Refactorizar separa los widgets con múltiples `StatefulWidget`s en lugar de la clase `StatelessWidget`. Significa que cada vez que el widget padre se redibuja, todas las clases de widgets no se redibujan. Se construyen una sola vez, lo que es genial para optimizar el rendimiento. El siguiente código de ejemplo muestra cómo usar una clase de widget para devolver un widget contenedor. Se inserta el widget `ContainerLeft()` de la `const` en el árbol de widgets donde sea necesario. Observe el uso de la palabra clave `const` para aprovechar el caching.

Ejemplo

```
class ContainerLeft extends StatelessWidget {  
  const ContainerLeft({  
    Key key,  
  }) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      color: Colors.yellow,  
      height: 40.0,  
      width: 40.0,  
    );  
  }  
}  
  
// Call to initialize the widget and note the const keyword  
const ContainerLeft(),
```

Veamos un ejemplo que refactoriza utilizando clases de widgets (un widget Flutter). Este enfoque mejora la legibilidad y el rendimiento del código al separar las partes principales del árbol de los widgets en clases de widgets separados. **¿Cuál es el beneficio de usar las clases de los widgets?** Es un rendimiento puro y simple durante las actualizaciones de pantalla. Cuando se llama a una clase de widget, es necesario utilizar la declaración `const`; de lo contrario, se reconstruirá cada vez, sin cachear. Un ejemplo de refactorización con una clase de widget es cuando tienes una disposición de la interfaz de usuario en la que sólo determinados widgets cambian de estado y otros permanecen igual.

La creación de un árbol de widgets poco profundo significa que cada widget está separado en su propia clase de widgets por su funcionalidad.

Ten en cuenta que la forma de separar los widgets será diferente dependiendo de la funcionalidad que se necesite.

En este ejemplo, creaste la clase de widgets RowWidget() para construir el widget Row horizontal con widgets hijos. La clase de widget RowAndColumnWidget() es un excelente ejemplo de cómo se puede aplanar aún más llamando a la clase de widget RowAndStackWidget() para uno de los widgets hijos de la columna. La clasificación por separado añadiendo la clase adicional RowAndStackWidget() se hace para mantener el árbol de widgets plano porque la clase RowAndStackWidget() construye un widget con múltiples hijos . En el código fuente del proyecto, añadí para su conveniencia un botón que aumenta un valor de contador, y cada clase de widget usa una declaración de impresión para mostrar cada vez que se llama a cada uno cuando cambia el estado del contador. Lo siguiente es el archivo de registro que muestra cada vez que un widget es llamado.

Cuando se toca el botón, el widget CounterTextWidget se redibuja para mostrar el nuevo valor del contador, pero observe que los widgets RowWidget , RowAndColumnWidget y RowAndStackWidget se llaman sólo una vez y no se redibujan cuando cambia el estado. Usando la técnica de la clase de widgets, sólo se llaman los widgets que necesitan ser redibujados, mejorando el rendimiento general.

```
// App first loaded
flutter: RowWidget
flutter: RowAndColumnWidget
flutter: RowAndStackWidget
flutter: CounterTextWidget 0

// Increase value button is called and notice the row widgets are not redrawn
flutter: CounterTextWidget 1
flutter: CounterTextWidget 2
flutter: CounterTextWidget 3
```

Refactorizar

A medida que el número de widgets aumenta, el árbol de widgets se expande rápidamente y disminuye la legibilidad y manejabilidad del código.

Para mejorar la legibilidad y manejabilidad del código, es posible separar los widgets en su propia clase, creando un árbol de widgets menos profundo.

En cada aplicación, es necesario hacer el esfuerzo por mantener el árbol de widgets poco profundo.

Al refactorizar con una clase de widget, se puede aprovechar la reconstrucción del subárbol de Flutter, que mejora el rendimiento.
