Desarrollando y diseñando aplicaciones móviles con Flutter

Decena, Facundo – Maguire, Margarita – Merenda, Franco – Pellegrino, Maximiliano

InforSanLuis 2019

Cronograma

- **Clase 1:** Introducción a Flutter y Dart
- **Clase 2:** Primera aplicación en Flutter
- **Clase 3:** Layouts y diseño de UI
- **Clase 4:** Evaluación basada en una pequeña aplicación

Requisitos

- Programación orientada a objetos (en cualquier lenguaje)
- Notebook (en lo posible) con Android Studio configurado para Flutter y Dart (lo vemos en clase).
- Para quienes tienen Android: cable USB

Introducción a Flutter y Dart

InforSanLuis 2019

¿Qué es Flutter?

Flutter es un **kit de desarrollo de software** (SDK) de código abierto desarrollado por Google.

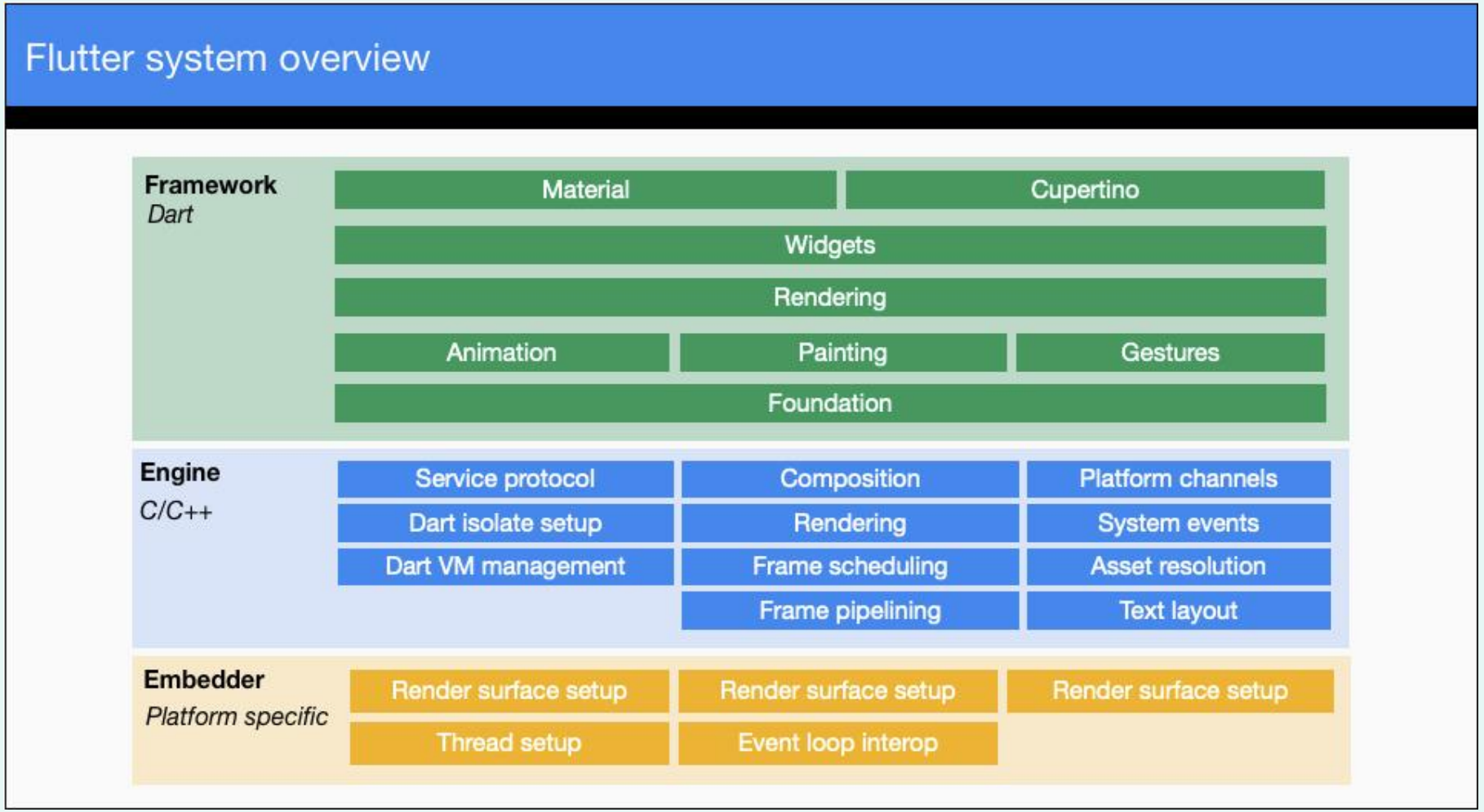
Puede usarse para desarrollo de aplicaciones **móviles** tanto para Android como para iOS, como para aplicaciones **web** y de **escritorio**.

Flutter utiliza **Dart** como lenguaje para desarrollo.

Arquitectura de Flutter

- **Plataforma de Dart:** Dart como lenguaje, compilación JIT en debug y AOT en release.
- **Flutter Engine:** Renderizado a bajo nivel. Comunicación con SDK específicos de la plataforma.
- **Liberías de Base:** Clases y funciones básicas escritas en Dart.
- **Widgets específicos de diseño:** Material y Cupertino.

Arquitectura de Flutter



¿Por qué usar Flutter?

- Productividad: Nos permite con un solo código desarrollar aplicaciones para diversas plataformas.
- Más con menos código.
- Prototipos e iteraciones fácilmente.
- Amplia variedad de Widgets (Material, Cupertino, custom widgets).

¿Por qué Dart?

- Varios criterios de selección para el lenguaje.
- Compilación JIT que permite **hot reload** durante el desarrollo de las aplicaciones, agilizando el trabajo.
- Compilación AOT que maximiza la **performance** de las aplicaciones con código nativo eficiente para ARM.
- Flexibilidad de la comunidad de Dart para adaptarse a las necesidades de Flutter

Dart Overview

¿Qué es Dart?

Es un lenguaje de programación multiplataforma de código abierto desarrollado por Google. Está inspirado en Java, JavaScript, Swift, etc.

Permite compilar nativo para ARM y x64, e incluso compilar a JavaScript para la web.

Está optimizado para el desarrollo de interfaces de usuario.

Es fácil de leer y de aprender, sintaxis clara y familiar.

Conceptos importantes

- Todo lo que se pueda guardar en una variable, es un **objeto.** Números, funciones, incluso null.
- Fuertemente tipado con inferencia de tipos usando la palabra reservada “var”.
- Soporta tipos genéricos al igual que Java (List<T>).
- Permite anidamiento de funciones.
- Variables de instancia = propiedades, campos

Conceptos importantes

- No existen modificadores de acceso public, private, protected, etc, como en Java. Si el nombre comienza con “_” es privado a la librería.
- Soporte para la asincronía.
- Posee expresiones condicionales y sentencias condicionales (if-else).

Conceptos importantes

Expresión condicional

```
var visibility = isPublic ? 'public' : 'private';
```

Sentencia condicional

```
if(isPublic){ visibility = 'public'; }
else{ visibility = 'private'; }
```

Variables

- Se pueden definir sin tipo, para que sea inferido

```
var nombre = "InforSanLuis";
```

- Cambio dinámico de tipo durante ejecución

```
dynamic nombre "InforSanLuis";
```

- Definición explícita del tipo de la variable

```
String nombre "InforSanLuis";
```

Variables

El valor por defecto de toda variable sin inicializar es **null**.

Final vs **Const:**

- **final:** Su valor se asigna una única vez y no se vuelve a cambiar.
- **const:** Es una constante en tiempo de compilación.

Se agregan a la definición de la variable, reemplazando el var o en conjunto con el tipo declarado

```
final nombre = "InforSanLuis";
final String nombre = "InforSanLuis";
```

Tipos

Los tipos proporcionados por Dart son:

- Números: clase: num, sub clases: int, double
- Strings
- Booleanos: true, false
- Listas: arreglos, homogéneos, usan corchetes.
- Conjuntos: homogéneos, sin repetidos, usan llaves.
- Mapas: clave : valor. Clave única.
- Etc.

Al ser objetos, algunos de estos tipos pueden ser inicializados con constructores. Ej, Map()

Funciones

Las funciones son objetos de tipo **Function**. Pueden ser asignadas a variables, pasadas como parámetros, retornadas.

```
int mayor(int primero, int segundo){
    if(primero > segundo)
        return primero;
    else
        return segundo;
}
```

Funciones de una sola línea con notación *arrow*, “flecha”

```
int mayor(int primero, int segundo) => primero > segundo ? primero : segundo;
```

Parámetros

- Parámetros posicionales opcionales

```
void nombre(String primerNombre, String apellido,  [String segundoApellido]){}
```

- Parámetros por defecto

```
void imprimir(String texto, [String color = "negro"]){}
```

Parámetros

- Parámetros nombrados

```
void contenedor({Color color, bool activado}){}

contenedor(color: Color.rgb(255,255,255), activado: true );
```

- Anotación para parámetro obligatorio

```
void contenedor({Color color, @required bool activado}){}
```

Funciones

- Permite funciones anónimas, con varias sentencias o de una sola línea.
- Las funciones son objetos de primer orden.
- Alcance estático de las variables.
- Permite anidamiento de funciones.
- Todas las funciones retornan algo, si no se explicita retorna null.

Operadores

Description	Operator
unary postfix	`expr++ expr-- () [] . ?.`
unary prefix	`-expr !expr ~expr ++expr --expr await expr`
multiplicative	`* / % ~/`
additive	`+ -`
shift	`<< >> >>>`
bitwise AND	`&`
bitwise XOR	`^`
bitwise OR	`|`

Operadores

relational and type test	`>= > <= < as is is!`
equality	`== !=`
logical AND	`&&`
logical OR	`||`
if null	`??`
conditional	*`expr1 ? expr2 : expr3`*
cascade	`..`
assignment	`= *= /= += -= &= ^=` *etc.*

Operador Cascada

Permite realizar una secuencia de operaciones sobre el mismo objeto. Evita la creación de variables auxiliares.

Con variable auxiliar

```
var button = querySelector('#confirm');
button.text = 'Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));
```

Usando cascada

```
querySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

Null-Aware

El operador null-aware nos permite hacer operaciones solo en caso de que un valor dado no sea nulo, simplificando las expresiones. Evita errores y produce código más simple.

```
var x;
if(y != null){
  x = y;
}
else{
  x = z;
}
```

```
var x = y ?? z;
```

Null-Aware

Algunos ejemplos utilizando el operador

```
// Asigna y a x solo si no es null, sino asigna z
var x = y ?? z;

// Asigna y a x solo si x es null
x ??= y;

// Llama a funcion() solo si x no es nulo
x?.funcion();
```

Estructuras de control

- If-else
- For loop
- For each
- For-in (usando el operador in)
- While
- Do-while
- Switch-case

Excepciones

Permite arrojar y capturar excepciones. Los programas Dart también pueden arrojar objetos no nulos, no solo excepciones.

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // A specific exception
  buyMoreLlamas();
} on Exception catch (e) {
  // Anything else that is an exception
  print('Unknown exception: $e');
} catch (e) {
  // No specified type, handles all
  print('Something really unknown: $e');
}
```

- Operador **on**: usado cuando se especifica el tipo de la excepción.
- Operador **catch**: usado cuando se necesita trabajar con el objeto de la excepción.
- Finally.

Clases

- La palabra **new** es opcional para crear instancias (se desaconseja su uso).
- Los getter se crean implícitamente para variables de instancia, y los setter si la variable no es final.
- Modificador **static** para atributos de clase.
- No se heredan los constructores de la superclase.
- Constructores nombrados para mayor claridad.
- Modificador **abstract** para métodos y clases.

Clases

Constructor nombrado

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

Declaración de getter y setter

```
class Rectangle {
  num left, top, width, height;

  Rectangle(this.left, this.top, this.width, this.height);

  // Define two calculated properties: right and bottom.
  num get right => left + width;
  set right(num value) => left = value - width;
  num get bottom => top + height;
  set bottom(num value) => top = value - height;
}

void main() {
  var rect = Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

Asincronía

Las operaciones asincrónicas permiten que el programa complete el trabajo mientras espera que termine otra operación. Aquí hay algunas operaciones asincrónicas comunes:

- Obtener datos a través de una red.
- Escribir en una base de datos.
- Lectura de datos de un archivo

Para realizar operaciones asincrónicas en Dart, puede usar la clase **Future**, y las palabras clave **await** y **async.**

Asincronía

- **Future:** Una instancia de esta clase representa el resultado de una operación asincrónica, puede tener dos estados *completo o incompleto.*

- **async:** Palabra clave para definir a una función como asincrónica. Va siempre antes del cuerpo de la función.

```
Future<void> ejemploAsincronia() async {  }
```

Asincronía

- **await:** Lo que dice es básicamente, ejecute esta función de forma asincrónica, y cuando termine, continúe con la siguiente línea de código. *Solo funciona en funciones async.*

```
Future<String>  createOrderMessage() async {
  var order = await fetchUserOrder();
  return 'Your order is: $order';
}
```

Comentarios

- Comentarios de una sola línea **//**
- Comentarios en bloque **/* */.** Todo lo que esté dentro es ignorado por el compilador.
- Comentarios de documentación **///** o **/****. Todo lo que esté dentro es ignorado salvo lo que se encuentre entre llaves. Lo que se encuentre entre corchetes se convierte en una referencia a la documentación de la clase con ese nombre. Ej: Crea un objeto [Persona].

Effective Dart

Effective Dart son las buenas prácticas para el lenguaje. Posee diferentes guías (Estilo, Diseño, Documentación y Uso) para generar código consistente y breve.

Cada guía posee una de las siguientes palabras:

- **DO:** prácticas a seguir siempre
- **DON'T:** prácticas que casi nunca son buena opción
- **PREFER:** prácticas que son aconsejables en la mayoría de casos, aunque a veces sea coherente no hacerlo.
- **AVOID:** prácticas que por lo general no se aconsejan, pero en algunos casos puede haber una razón válida
- **CONSIDER:** prácticas a seguir según preferencia, necesidad, etc.

Effective Dart

Estilo:

- **DO** nombrar tipos usando UpperCamelCase.
- **DO** nombrar librerías, paquetes, directorios con lowercase_with_undercores.
- **DO** nombrar identificadores usando lowerCamelCase.
- **DO** usar llaves para todas las sentencias de control.

Documentación:

- **DON'T** no usar comentarios en bloque para documentar.
- **PREFER** ser breve.

Effective Dart

Uso:

- **DO** usar strings adyacentes para concatenar literales de string.
- **PREFER** usar interpolación para componer strings y valores.
- **DON'T** no usar .length para ver si una colección está vacía.
- **DON'T** no inicializar variables explícitamente con null.

Diseño:

- **AVOID** parámetros posicionales booleanos.
- **AVOID** declaración de tipo en variables locales inicializadas.
- **DO** utilizar términos consistentemente para los nombres.

Effective Dart

Tour de Dart completo

- https://dart.dev/guides/language/language-tour

Lista completa de guías de Effective Dart

- https://dart.dev/guides/language/effective-dart

Getters y Setters

- https://stackoverflow.com/questions/27683924/how-do-getters-and-setters-change-properties-in-dart

Flutter

UI Declarativa

El enfoque declarativo de Flutter hace más sencilla la transición entre estados de interfaz. En lugar de construir la interfaz y luego modificarla mediante métodos, se describe el contenido actual y el framework se encarga de las transiciones.

```
// Imperative style
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

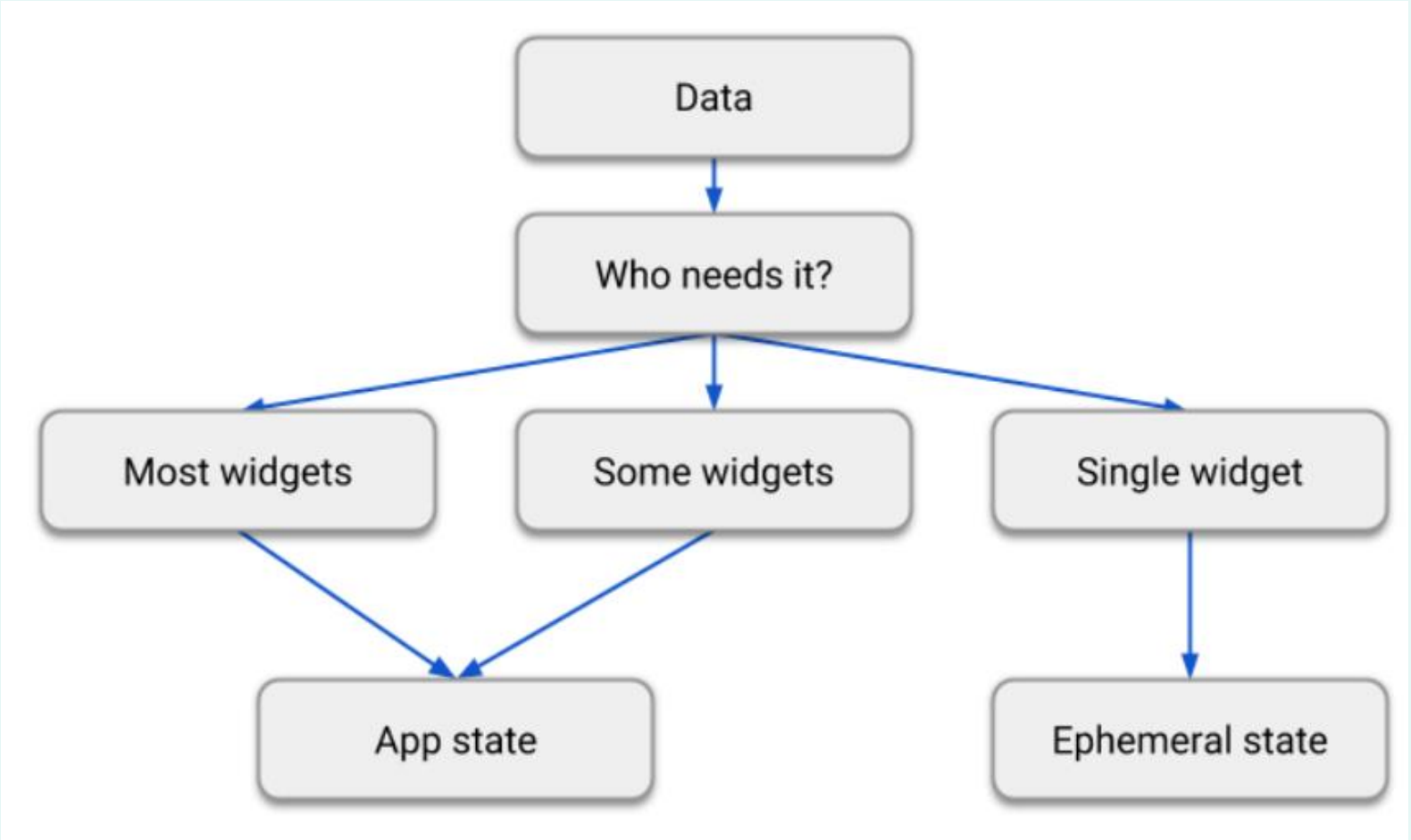
```
// Declarative style
return ViewB(
  color: red,
  child: ViewC(...),
)
```

Estados

Un estado son todos los datos que se necesitan para redibujar la interfaz en algún momento del tiempo. Se pueden diferenciar en dos tipos conceptuales:

- **Estados efímeros:** El estado de un único widget. No se necesitan técnicas o arquitecturas para manejarlos. No es necesario que persistan durante mucho tiempo.
- **Estados de aplicación:** Es un estado que se desea compartir con muchas partes de la aplicación, y que se quiere compartir entre sesiones de usuario. Ejemplos: información de loggeo, el carrito de compras de una web, leídos/no leídos, etc.

Estados


Data
Who needs it?
Most widgets
Some widgets
Single widget
App state
Ephemeral state

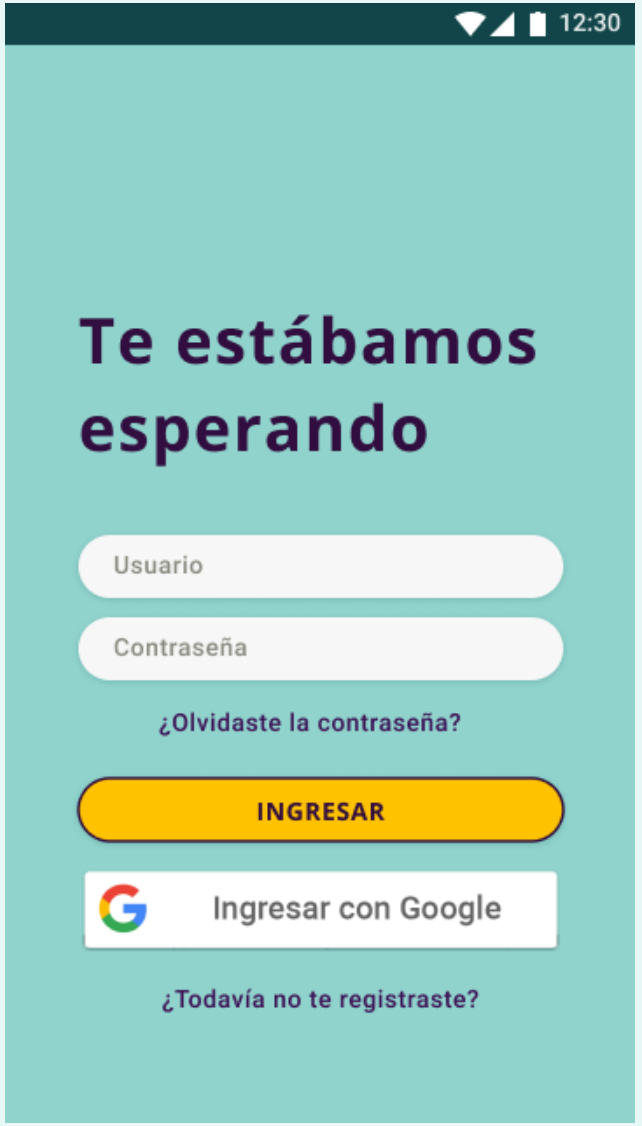
Widgets

En Flutter todo es un Widget. Más específicamente, un Widget es una descripción inmutable de una parte de la interfaz de usuario. Nos describen cómo debe verse la vista en función de la configuración actual y del estado de la aplicación.

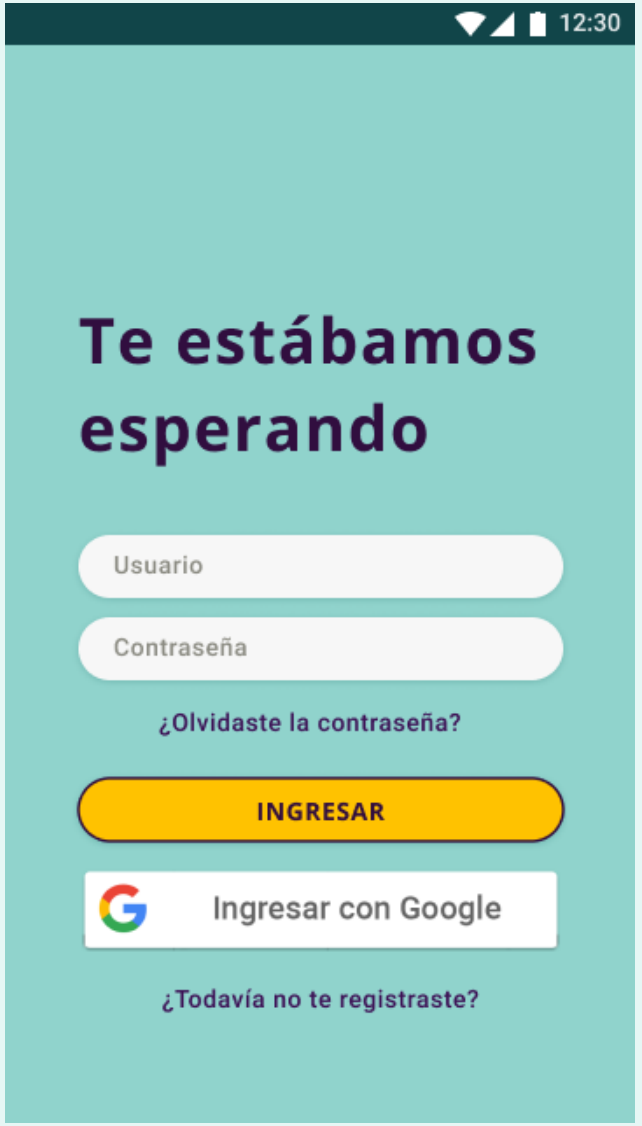
Cuando el estado de un widget cambia, reconstruye su descripción, y el framework lo compara con la descripción anterior para determinar los mínimos cambios necesarios en el árbol de renderizado para pasar de un estado al próximo.


12:30
Reuniones
Fiestas
Eventos
Juntadas
Todo en un solo lugar
Sumate a meeti
EMPEZAR
INGRESAR

VS


12:30
Te estábamos esperando
Usuario
Contraseña
¿Olvidaste la contraseña?
INGRESAR
Ingresar con Google
¿Todavía no te registraste?

VS



Stateless

Stateful

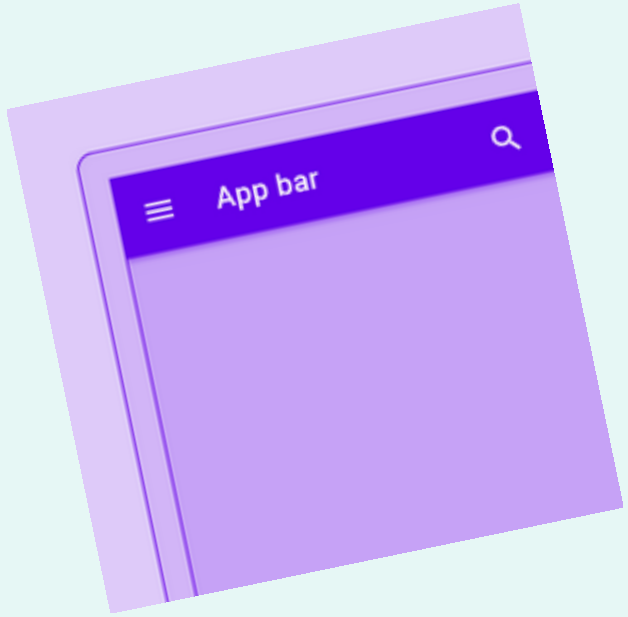
Widgets

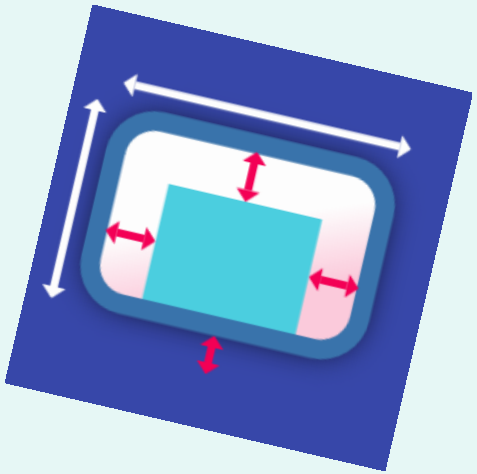
- **Stateless:** No necesitan un estado mutable. Son útiles cuando la parte de la interfaz que se está describiendo no depende de nada más sino de la información propia del widget y del contexto en el cual se dibuja.
- **Stateful:** Posee un estado mutable. Cuando algún valor o información sobre el widget se modifica durante su tiempo de vida, se debe notificar para que se vea reflejado. Son útiles cuando la parte de la interfaz que se está describiendo puede cambiar dinámicamente en función de estados de reloj o por estados de la aplicación misma.

Widgets

Categorías

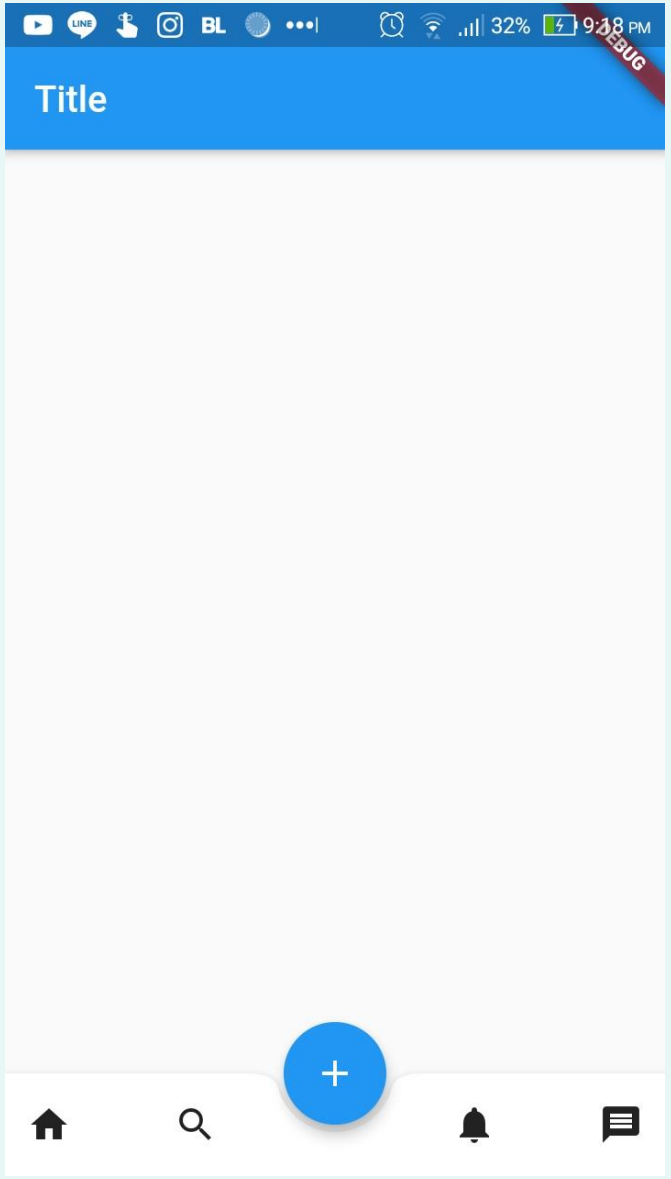
- Básicos
- Input
- Layout
- Texto
- Estilo
- Imágenes
- Animación
- Scrolling
- Etc.





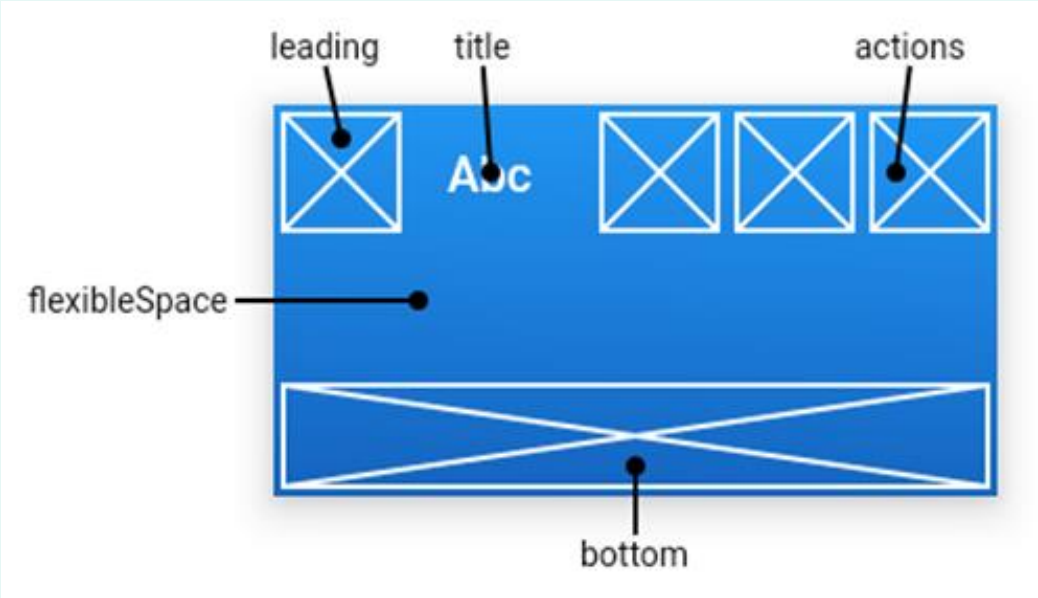
Scaffold

Es la estructura de diseño básica de Material design. Nos provee una barra superior (AppBar), una barra inferior (BottomNavigationBar), un cuerpo para nuestro diseño de la pantalla, un botón flotante (FloatingActionButton), entre otras cosas. Permite también mostrar drawers, snack bars, etc.



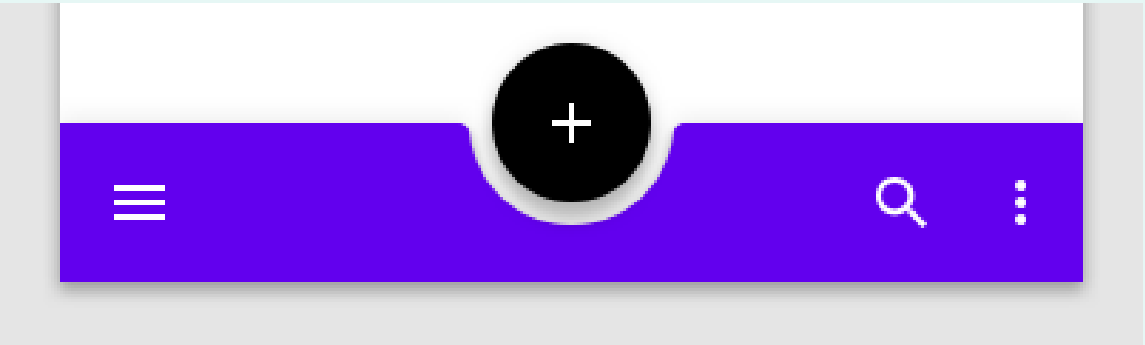
AppBar

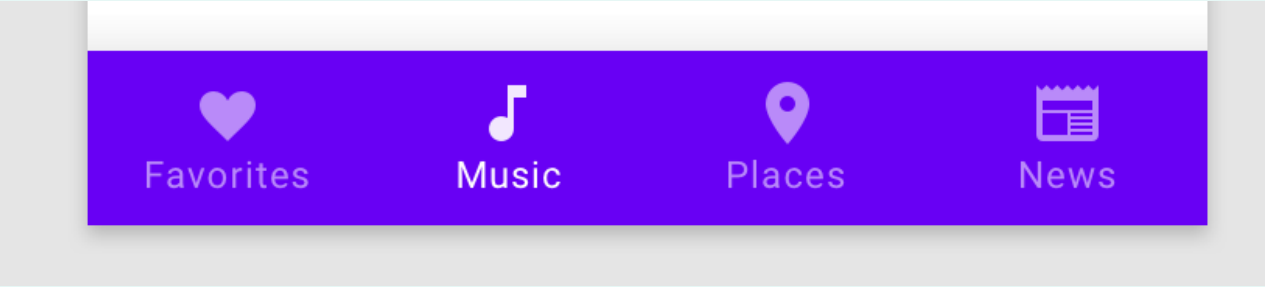
Es una barra superior de material. Es una barra de tareas que puede contener una tab bar. Poseen generalmente acciones con íconos, y un menú desplegable para acciones menos comunes.



BottomNavigationBar

Se muestra al fondo de una aplicación, junto con un conjunto de íconos, textos, o ambos, que proveen navegación rápida a través de la aplicación.



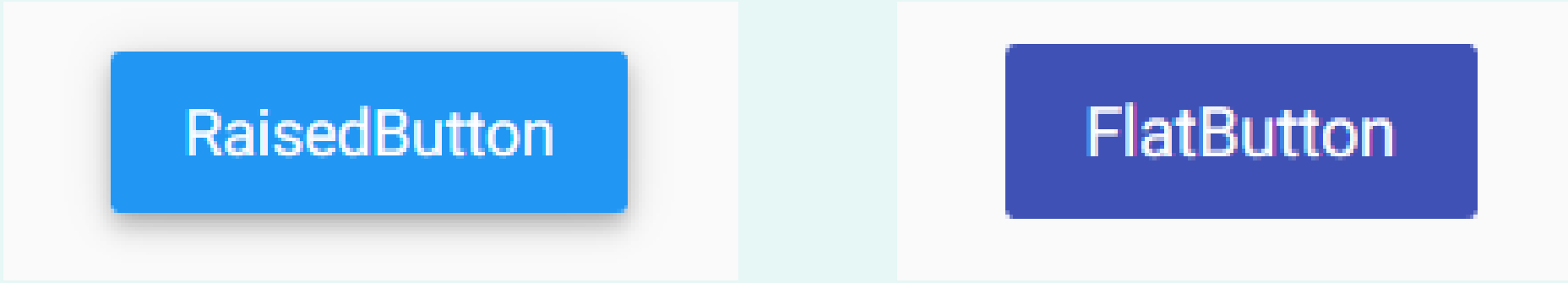


RaisedButton y FlatButton

El RaisedButton es un botón de Material con elevación, utilizado en diseños planos.

El FlatButton es un botón de Material con elevación 0.

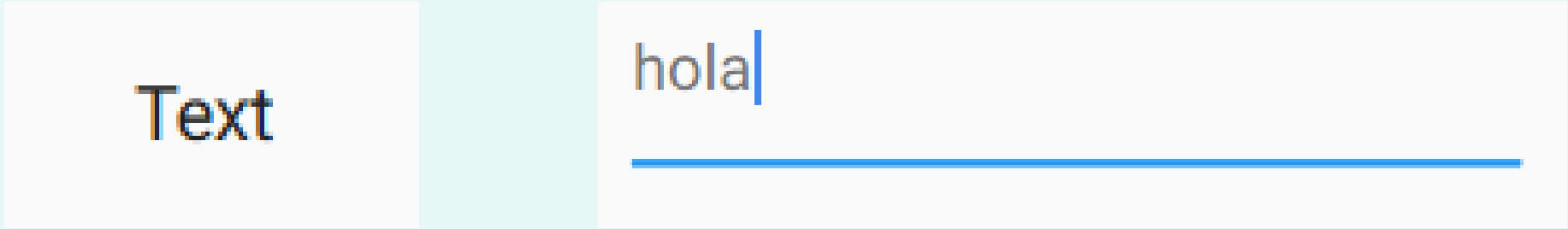
Ambos pueden configurar acciones para cuando son presionados y personalizar su apariencia.



Text y TextField

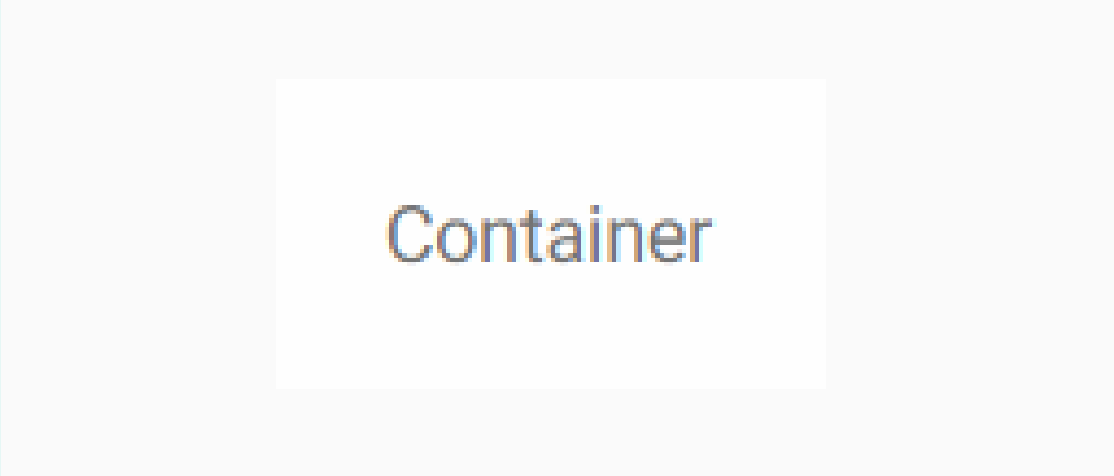
El Text es un widget que muestra un string de texto con un estilo. Puede mostrarse en una o más líneas en función de las restricciones.

El TextField es un campo que le permite al usuario ingresar texto, en teclado real o en pantalla.



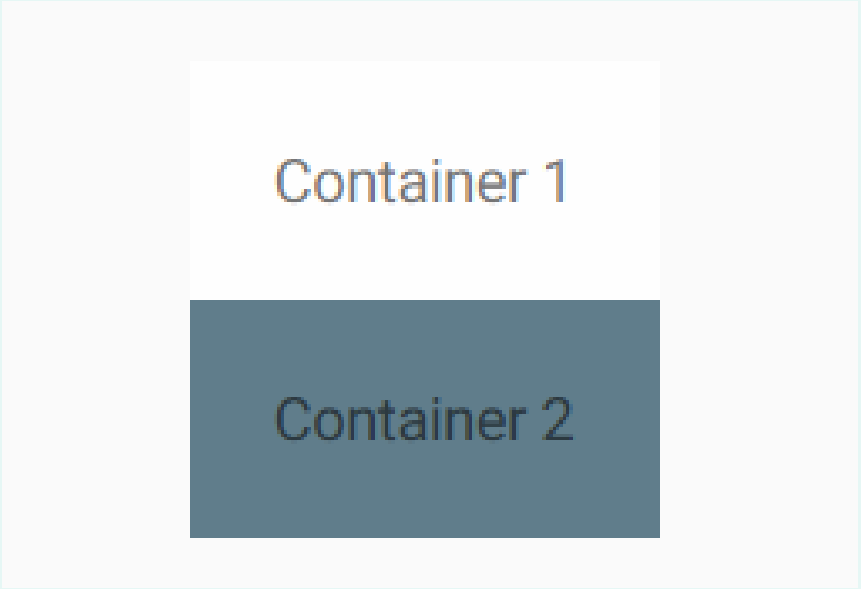
Container

Es un widget que envuelve un widget, proporcionándole un color de fondo, posición, tamaño, etc. Le da restricciones al widget contenido en él, padding, y más.



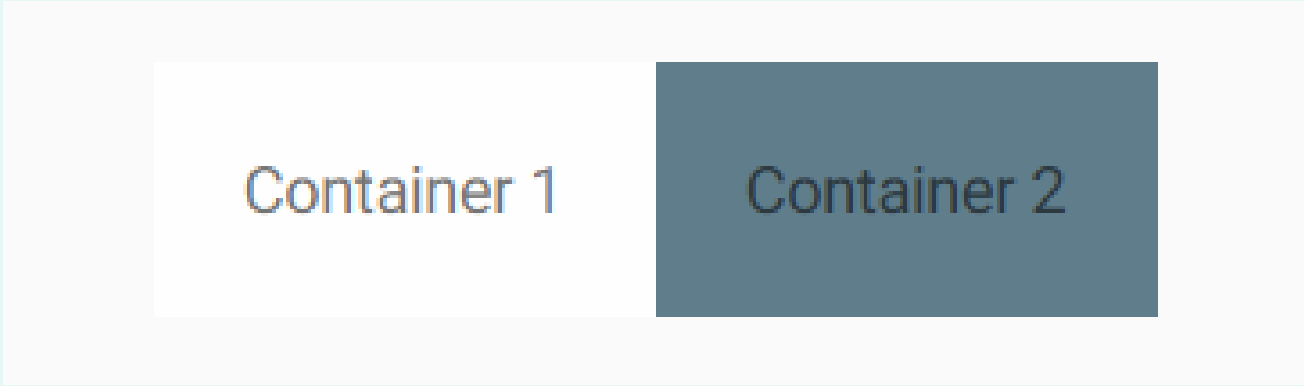
Column

Es un widget para layout que nos permite organizar otros widgets en la pantalla. Nos ubica uno debajo del otro, con diferentes alineaciones que deseemos. Vertical.



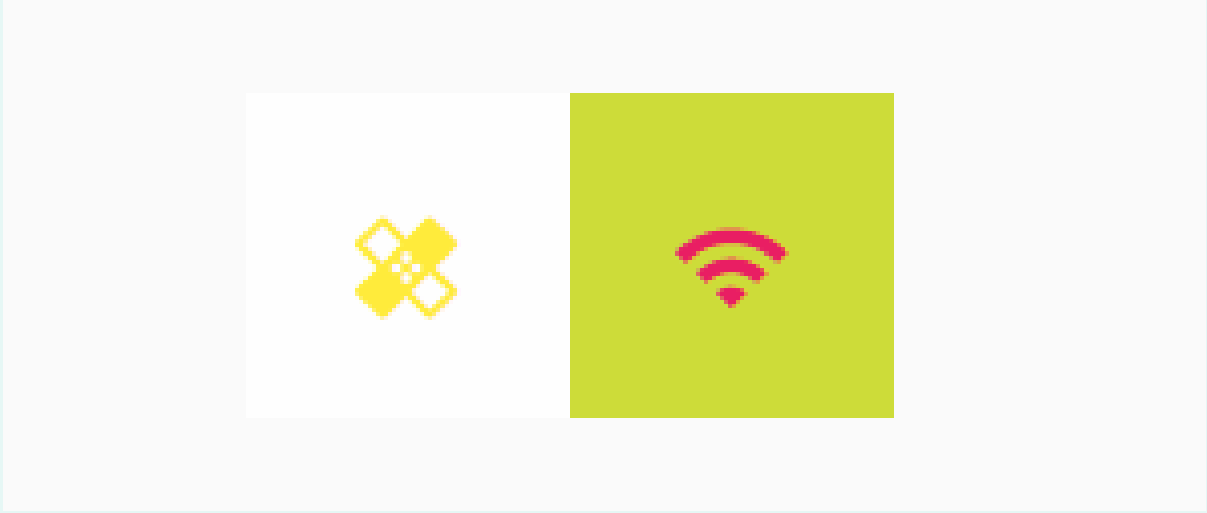
Row

Es un widget para layout que nos permite organizar otros widgets en la pantalla. Nos ubica al lado del otro, con diferentes alineaciones que deseemos. Horizontal.



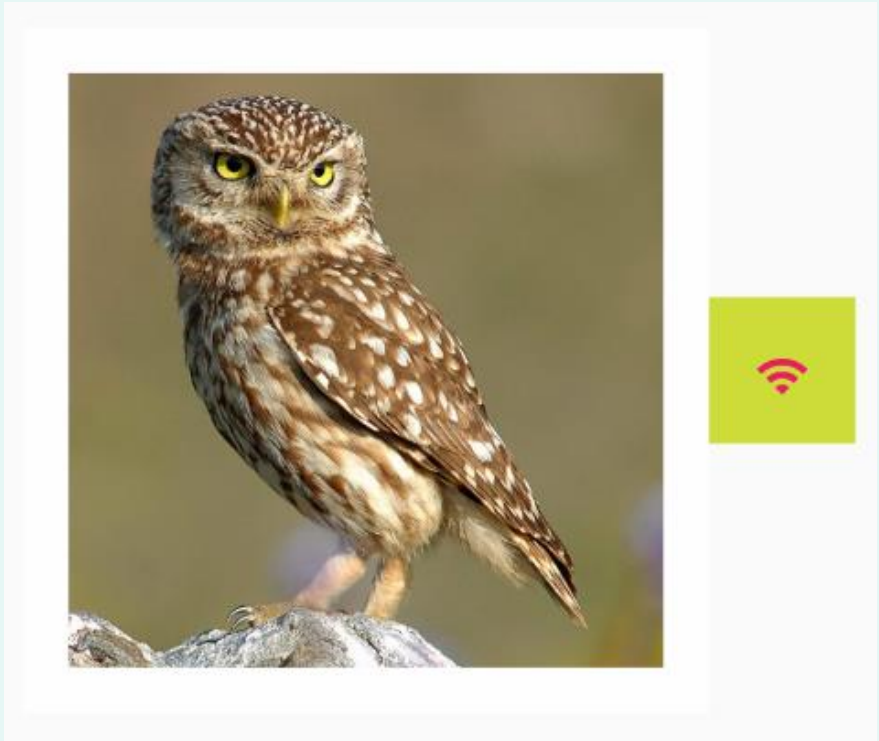
Icon

Dibuja un ícono en base a cómo esta descripto en IconData. Ejemplo, la clase Icons de Material. Renderizan íconos cuadrados.



Image

Nos sirve para mostrar una imagen. Posee varios constructores nombrados en función de desde dónde obtiene la imagen.



Widgets

Catálogo de Widgets por categorías

- https://flutter.dev/docs/development/ui/widgets

Índice de Widgets alfabéticamente

- https://flutter.dev/docs/reference/widgets

Widget of the week, lista de videos de Youtube con explicación y ejemplos en código (en inglés)

- https://www.youtube.com/playlist?list=PLjxrf2q8roU23XGwz3Km7sQZFTdB996iG