## **Flutter. Primeros pasos**

1. [¿Qué es Flutter?](#kix.nr1oxmdgrnq6) 1
2. [Instalación](#kix.s9r6tulqqqop) 2
3. [Estructura de un proyecto en Flutter](#kix.dvo2u8o5do8p) 5
4. [Nuestras primeras apps en Flutter](#kix.v3g8l5hi9g2n) 7

4.1 [Hola mundo](#kix.8of0fnfzt2k) 7

4.2 [Introducción de una barra superior](#kix.7kfm8o81bmw3) 8

[¿Qué son los widgets?](#kix.l1ufgrczro7w) 11

4.3 [Insertar una imagen](#kix.t1shu01v58vi) 12

1. [Estilos de texto](#kix.do7dpchm0mos) 13

5.1 [Textos con estilo único](#6huwrem1kk8s) 13

5.2 [Textos con varios estilos](#g3d7k7i13our) 26

5.3 [Incluir fuentes de Google Fonts](#_5aii7hxub5k5) 28

1. [Trabajar con iconos](#_d9yqrv1i44xh) 30
2. [Disposición de elementos. Row y Column](#paji1v5l17gg) 31
3. [Widget Scaffold. Ampliación. Drawers](#p500mvyrsoav)32

8.1 [División del código fuente](#37l7q6moi1pf) 33

8.2 [Navegación entre pantallas](#at4bie712ed) 35

8.3 [Drawer](#ng3z4sxkl3g) 36

1. [Finalización de una apk con flutter](#tnapqwzfbypp) 42

## **¿Qué es Flutter?**

## [**Flutter**](https://flutter.io/) es una herramienta que ofrece Google para poder construir aplicaciones que se puedan ejecutar en distintas plataformas, como Android o iOS, con un solo código base. Permite construir interfaces complejas, peticiones a la red, trabajar con ficheros... Flutter no funciona como otros frameworks multiplataforma como puede ser Cordova. A diferencia de Cordova, Flutter no se ejecuta sobre un WebView ejecutando HTML, CSS y Javascript. Flutter utiliza solo Dart como lenguaje de programación, un lenguaje bastante sencillo de aprender y si es el caso y sabes Java puedes aprender el Dart en poco tiempo.

Pero nuestra app no ejecutará el código en Dart sino que cuando compilamos nuestra app se compila a código nativo por lo que obtenemos un mayor rendimiento y una mejor respuesta de la interfaz de usuario.

Flutter es un framework de desarrollo móvil para crear aplicaciones nativas en Android e iOS. Tiene las siguientes características:

### **1.1 Desarrollo híbrido**

Flutter permite generar aplicaciones que se ejecutan tanto en iOS como en Android. Estas aplicaciones comparten el mismo desarrollo y librerías lo que las simplifica aún más. Para asegurarse de que la aplicación se "siente" igual en cualquier plataforma, Flutter utiliza *widgets* con estilos definidos por Cupertino (iOS) y Material Design (Android), además otros *Plugins de Flutter* que brindan acceso a servicios específicos de cada plataforma y API's.

### **1.2. Hot reload**

Permite recargar la aplicación en tiempo real desde el editor de código tanto en emuladores como en dispositivos físicos. Esto evita el tiempo que se necesita para poder realizar pruebas, por ejemplo generando un nuevo APK. Flutter permite el inyectar cambios en bloques a una máquina virtual de Dart, que se está ejecutando en el dispositivo, mediante el hot reload, Flutter permite ver los cambios en menos de 1 segundo. Algo muy importante de notar es que el *estado* de la aplicación se mantiene, así que no es necesario que se regenere manualmente. Por ejemplo si la aplicación tiene una pantalla de login, no será necesario el loguearse de nuevo en la aplicación. Esto es cierto siempre y cuando no se cambie algo relacionado con el estado de la aplicación.

### **1.3. Widgets**

Básicamente cada componente dentro de la aplicación son widgets, lo que permite personalizar el despliegue tanto como sea necesario.

### **1.4. Rendimiento nativo**

La aplicación generada es de código nativo, por lo tanto el rendimiento es muy similar a una aplicación directamente nativa.

## **1.5. Dart**

Flutter utiliza el lenguaje de programación Dart, que es de código abierto y desarrollado por Google. El propósito inicial era ofrecer una alternativa más moderna a *JavaScript*. *Lars Bak* (Google) lo define como un *"lenguaje estructurado pero flexible para programación Web"*.

## **2. Instalación**

Para la instalación es diferente para cada sistema operativo:

## [**Install | Flutter**](https://docs.flutter.dev/get-started/install)

## El proceso de instalación de Flutter lo podemos ver actualizado en la página oficial:

## [**Make Android apps | Flutter**](https://docs.flutter.dev/get-started/install/windows/mobile)

## **Pasos para la instalación:**

## En la versión actual, la instalación se resume en los siguientes pasos:

## 1º Instalar **git**. Este programa no requiere configuraciones adicionales.

## 2º Descargar **sdk de Flutter** de la página oficial. Extraerlo en c:\flutter. En la página oficial recomienda que se extraiga en c:\src\flutter. No es conveniente instalarlo en archivos de programa.

## 3º Añadir variable de entorno a Windows c:\flutter\bin (o c:\src\flutter\bin). Para ello hay que acceder a buscar teclear “env” y añadirlo al Path.

## 4º Una vez hecho esto, podemos ejecutar “flutter doctor” y nos dirá lo que le queda pendiente para que funcione correctamente

## 5º Instalar **Android Studio**. En la sección Extensiones, buscar flutter y añadirla. En “File>Settings”, acceder a “System Settings” y en la sección de arriba “SDK Tools” descargar “Android sdk Command-line Tools”.

## 6º Instalar **Visual Studio**. Una vez finalizada la instalación, acceder a “Herramientas>Acceder a herramientas y características…” e instalar “Desarrollo para el escritorio con C++”.

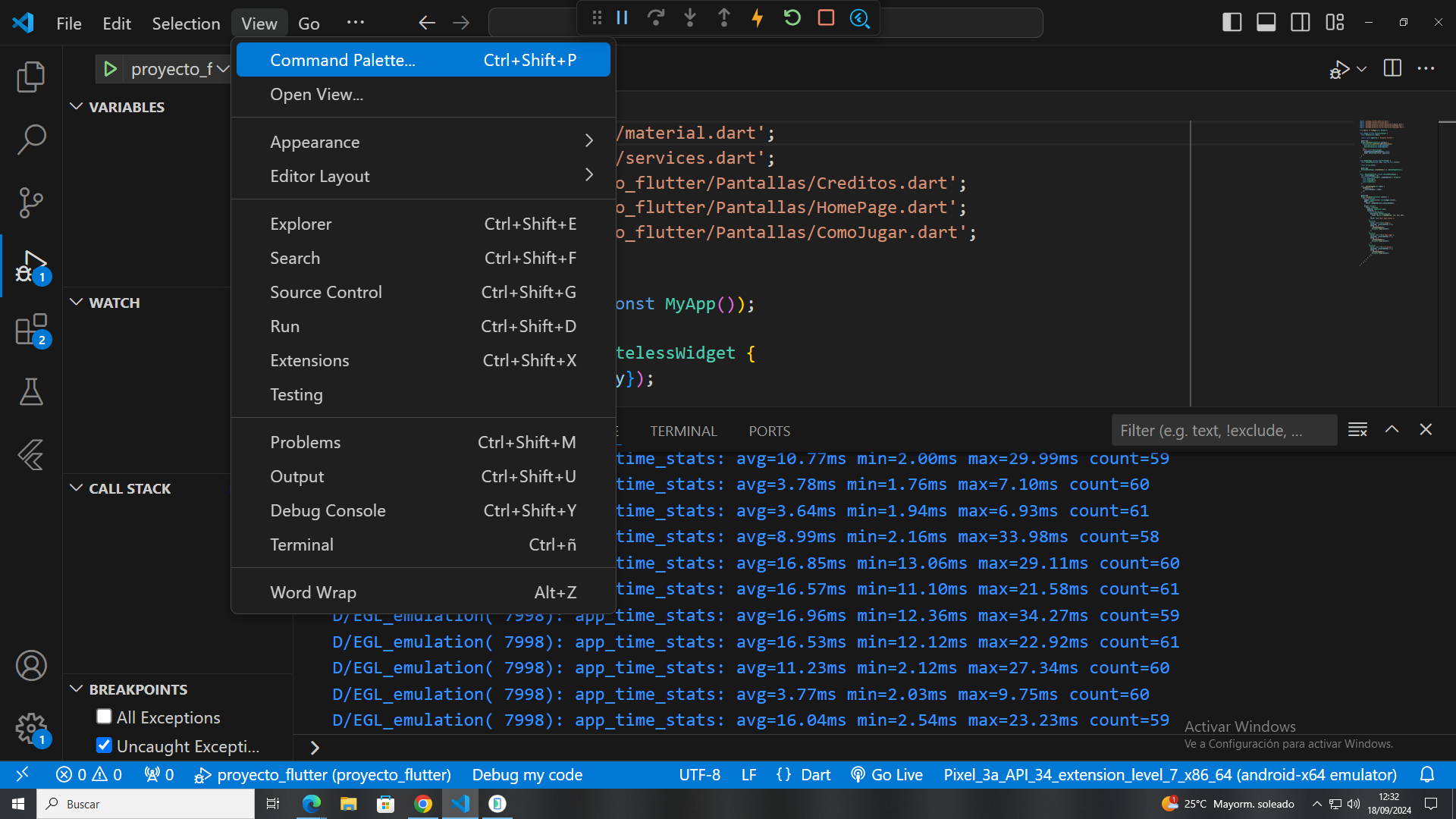
## 7º En línea de comandos ejecutar “flutter doctor –android-licences” y aceptar todas las licencias pulsando “y”.

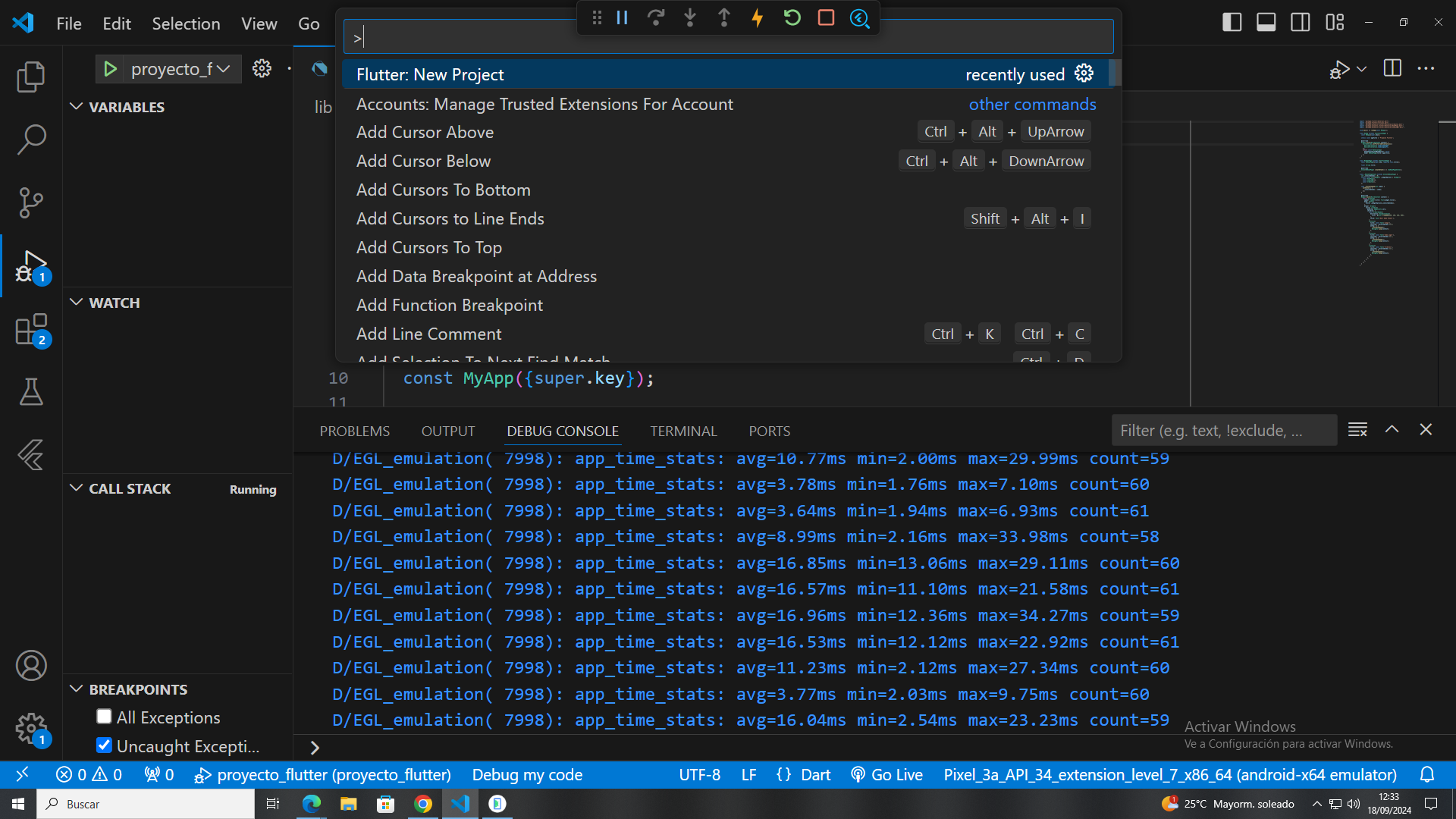
## 8º Instalar Visual Studio Code y las extensiones Flutter, **Awesome Flutter Snippets** y **Flutter Widget Snippets**. Estos dos últimos no son necesarios para la instalación, pero se utilizarán posteriormente para desarrollar código.

## **Comprobar que está bien instalado:**

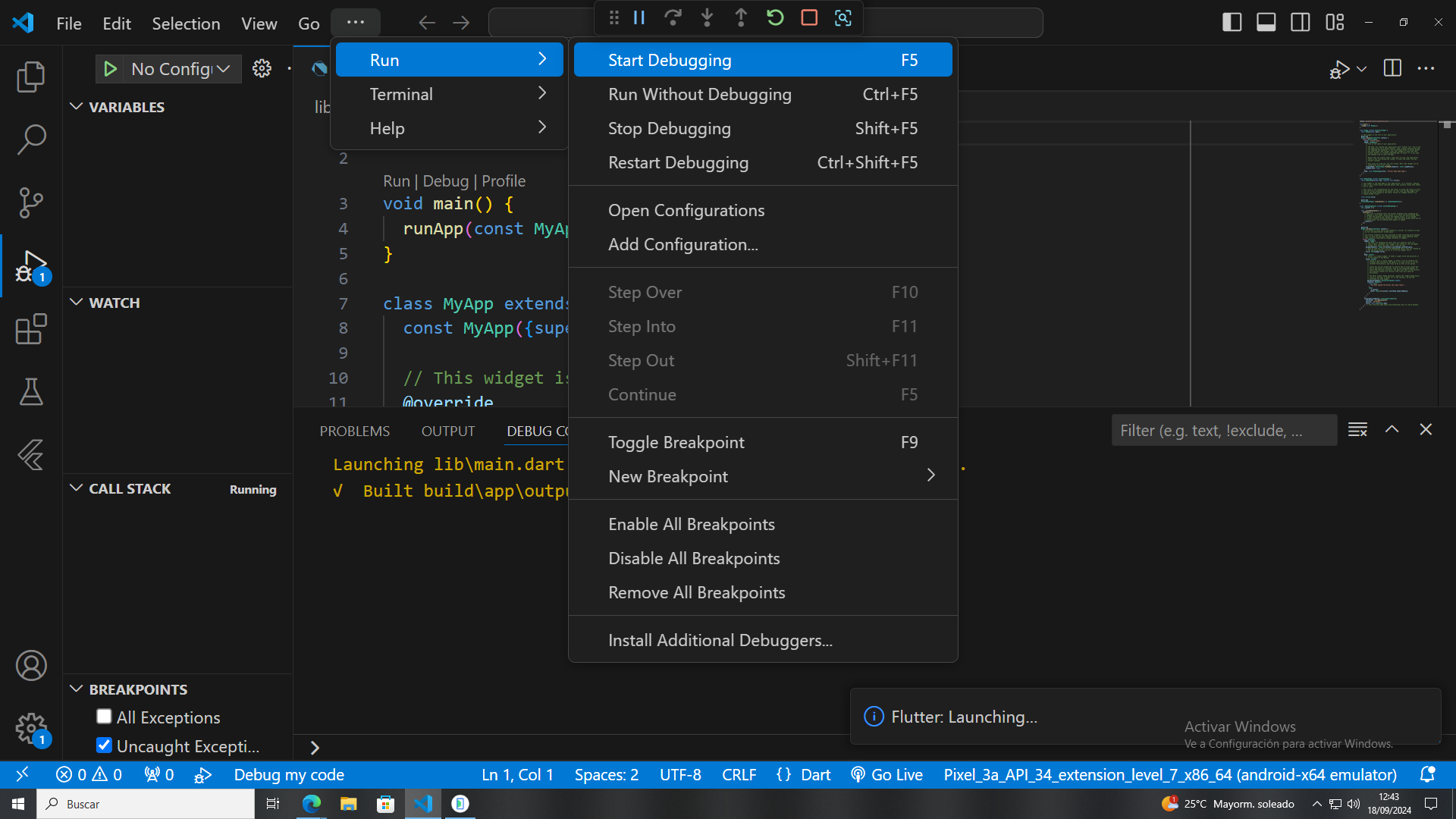
## Para comprobar que la instalación es correcta, habría que ejecutar el comando **“flutter doctor”** en la interfaz de comandos de Windows y que no dé errores.

Una vez instalado, podemos acceder a Visual Studio Code y acceder a “View>Command Palette…”

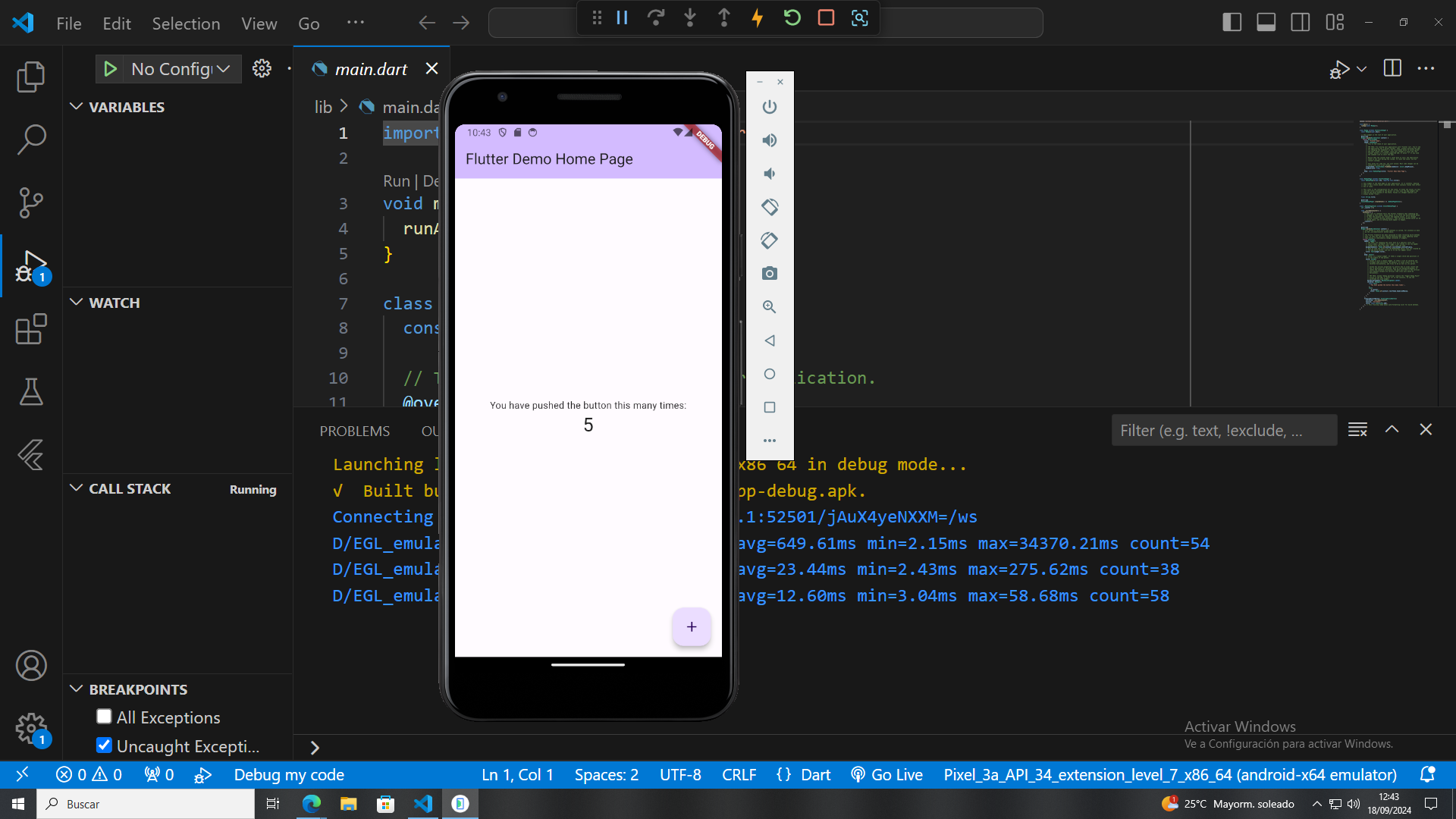


y pulsar en “flutter:New Project”, seleccionamos la carpeta y, por defecto flutter nos genera un programa de ejemplo que cuenta clics.

Para ejecutar la app, tendremos que acceder a “Run > start debuggin…” y nos lanzará el proyecto de flutter al emulador que hayamos instalado en Android Studio.



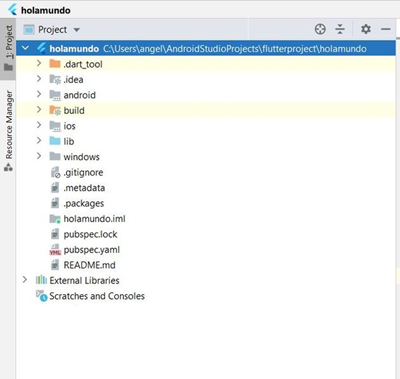
una vez hecho esto sale la app en el emulador:



## **3.** **Estructura de un proyecto Flutter**

## A continuación se indica a nivel general para que se usa cada una de las carpetas que aparecen por defecto al crear un nuevo proyecto Flutter.

## También existen directorios que no aparecen por defecto pero que serán necesarios crearlos para hacer nuestras aplicaciones.



## Podemos ver las siguientes carpetas:

## **idea**: Usado para que Android Studio guarde su configuración para el proyecto en curso. Podéis comprobar como dicho directorio no es tenido en cuenta si se sube a un repositorio git abriendo el archivo .gitignore.

## **android**: En esta carpeta se genera todo lo necesario para crear una aplicación Android a partir del proyecto Flutter en lenguaje Java / Kotlin. Cuando se instala la aplicación en el emulador, lo que hace Android Studio es generar el código necesario en formato 'nativo' de Android dentro de esta carpeta. Normalmente no será necesario modificar nada en esta carpeta.

## **ios**: En esta carpeta se genera todo lo necesario para crear una aplicación IOS a partir del proyecto Flutter. Normalmente no será necesario modificar nada en esta carpeta.

## **build**: Carpeta donde se genera el código al compilar una aplicación Flutter. Es donde se guarda la aplicación android apk generada. No es necesario modificar nada en esta carpeta.

## **lib**: Donde se encuentran normalmente los archivos que conforman el código fuente de la aplicación Flutter. Se puede estructurar como si fuera un sistema de archivos y permite crear directorios y subdirectorios.

## **test**: Carpeta donde se guardan los archivos para realizar pruebas sobre la aplicación de Flutter.

## También podemos visualizar los siguientes archivos:

## **.metadata**: Administrado por Flutter automáticamente y se usa para controlar las propiedades del proyecto Flutter, actualizaciones... Este archivo solo lo cambia Flutter y no debe ser modificado manualmente.

## **.gitignore:** Indica todos los archivos y directorios que deben ser ignarados en caso de subir a un repositorio git nuestro proyecto.

## **.packages**: Metadatos del proyecto Flutter. Contiene la lista de dependencias del proyecto. Es generado automáticamente por el SDK de Flutter y no debe ser modificado.

## **.pubspec.lock**: Indica cómo se construye el archivo pubspec.yaml. No debe editarse.

## **.pubspec.yaml**: Archivo que permite:

## Establecer la configuración general del proyecto como nombre, descripción y versión del proyecto

## Indicar las dependencias del proyecto (cuando hacemos uso de una nueva funcionalidad, por ejemplo)

## Indicar archivos de recursos, como imágenes, audios,...

## Tiene en cuenta la tabulación.

## Cuando se produce un cambio es necesario ejecutar el comando flutter pub get (o flutter packages get) (en casi todos los IDE´s aparecerá un mensaje indicándolo y ya lo ejecutará el propio IDE). Normalmente, para que se apliquen los cambios realizados en este archivo, necesitamos hacer un full restart, parando la aplicación y volviéndola a ejecutar para que la reinstale.

## **nombreproyecto.iml**: Este archivo siempre se nombra de acuerdo con el nombre del proyecto y contiene más configuraciones del proyecto Flutter. No debe modificarse.

## **README.md**: Archivo en el que aparece la descripción del proyecto. Es el archivo que aparece en un alojamiento de github en Internet.

## **4.** **Nuestros primeras apps en flutter**

## **4.1 Hola mundo**

El programa más básico que se suele realizar cuando aprendemos un lenguaje de programación es el siguiente:

| **import 'package:flutter/material.dart'**;  **void** main() {  runApp(  Text(**'Hola mundo!!!'**, textDirection: TextDirection.**ltr**)  );  } |
| --- |

En este programa nos encontramos con las siguientes instrucciones:

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

## Importamos la librería **material.dart** donde se encuentra definida **runApp()** y todos los widgets de flutter más habituales. Entre ellos el widget **Text**.

* **void** main() {}

Es la función principal de flutter.

* runApp()

Es el widget raíz del árbol de widgets que se muestra por pantalla.

* Text(**'Hola mundo!!!'**, textDirection: TextDirection.**ltr**

Utiliza el widget Text para visualizar el texto entre comillas por pantalla. A este widget hay que indicarle la dirección del texto para que se visualice de forma correcta, en este caso left to right (ltr).

## Mejorando un poco el anterior, vamos a introducir el widget Center y centramos el texto en el dispositivo:

| **import 'package:flutter/material.dart'**;  **void** main() {  runApp(  Center(  child: Text(**'Hola mundo!!!'**, textDirection: TextDirection.**ltr**)  )  );  } |
| --- |

Para ello, hacemos uso del widget **Center**, el cual centra el texto que se encuentra en **child**

## **4.2 Introducción de una barra superior**

## El siguiente ejemplo vamos a poner una barra superior y un botón predefinido:

| **import 'package:flutter/material.dart'**;  **void** main(){  runApp(  MaterialApp(  title: **'Mi primera aplicación'**,  home: Scaffold(  appBar: AppBar(  title: Center(  child: Text (**'Barra superior'**),  ),  ),  body: Center(  child: Text (**'Parte principal'**),  ),  floatingActionButton: FloatingActionButton(  child: Icon(Icons.*add*),  onPressed: (){},  ),  ),  )  );  } |
| --- |

## Para ello, hace uso de dos widgets: **MaterialApp** y **Scaffold**. Flutter nos proporciona el widget MaterialApp que encapsula diversos widgets utilizados en Material Design, que es una normativa de diseño enfocado en la visualización del sistema operativo Android, además en la web y en cualquier plataforma. Fue desarrollado por Google y anunciado en la conferencia Google I/O celebrada el 25 de junio de 2014. Ampliando la interfaz de tarjetas vista por primera vez en Google Now.

## **MaterialApp**

## Material es un lenguaje de diseño visual el cual es un estándar en web y móvil. Flutter ofrece un gran conjunto de “Material widgets”. Para utilizarlo, en la llamada al método runApp creamos una nueva instancia y se lo pasamos de la siguiente forma.

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

## void main() {

## runApp(

## MaterialApp(

## title: 'Flutter Demo',

## theme: ThemeData(

## primarySwatch: Colors.pink

## ),

## home: ...

## )

## );

## }

## El widget **MaterialApp** tiene como atributos entre otros:

## **title:** el título de la app.

## **theme:** el tema que vamos a usar, este atributo a su vez contiene otros atributos que podemos configurar. Como pueden ser **primarySwatch**, **brightness**, **primaryColor**, **accentColor…** En este ejemplo, vamos a especificar solo el primarySwatch con la ayuda de la clase Colors, que implementa internamente las especificaciones de color de material Design. Al especificar Colors.pink, que es un mapa con los colores para el rosa, al primarySwatch estamos definiendo la paleta de colores “pink” a la app.

## **home:** especificamos la primera ‘pantalla’ que queremos mostrar.

## **Scaffold**

## En Flutter el uso de Scaffold es muy importante y se utiliza con mucha frecuencia. Es un widget que es usado para contener otros widgets. Su constructor es el siguiente:

## Scaffold({

## Key key,

## this.appBar,

## this.body,

## this.floatingActionButton,

## this.floatingActionButtonLocation,

## this.floatingActionButtonAnimator,

## this.persistentFooterButtons,

## this.drawer,

## this.endDrawer,

## this.bottomNavigationBar,

## this.bottomSheet,

## this.backgroundColor,

## this.resizeToAvoidBottomPadding = true,

## this.primary = true,

## })

Las propiedades más utilizadas son:

* **appbar**: Es la barra superior.
* **body**: Es el cuerpo principal del widget.
* **floatingActionButton:** Muestra un botón en la parte inferior de la pantalla. Se puede personalizar y se utiliza si existe una acción principal clara. Un ejemplo lo tenemos en el código anterior.
* **persistentFooterButtons:** Si estableces este parámetro, mostrará estos widgets al pie del Scaffold. Este parámetro es un List.
* **drawer:** Dibuja un menú lateral de opciones a la izquierda de la pantalla.
* **bottomNavigationBar:** Proporciona una barra en la parte inferior de la pantalla con iconos y / o texto en los que el usuario puede hacer clic para moverse entre partes de la aplicación, entre otras cosas.
* **bottomSheet:** Este es un cuadro de diálogo que se muestra en la parte inferior de la pantalla pero sobre la bottomNavigationBar y los persistentFooterButtons.
* **backgroundColor & primary:** El backgroundColor cambia el color de fondo. El parámetro primary hará que el padding-top del Scaffold con la barra de estado sea cero. Cuando establezca primary = false, la pantalla se desplazará hacia arriba.

| Podemos encontrar más información respecto al widget Scaffold en:  <https://medium.com/comunidad-flutter/flutter-widgets-08-scaffold-5455cd1714b5>  <https://api.flutter.dev/flutter/material/Scaffold-class.html> |
| --- |

# **¿Qué son los Widgets?**

## Los widgets de Flutter son los distintos componentes que nos encontramos cuando construimos la interfaz de usuario (UI), toma inspiración de React. La idea central es que construyas tu UI de widgets y describen cómo debería ser su vista, dada su configuración y estado actuales. Cuando el estado de un widget cambia, el widget reconstruye su descripción, para determinar los cambios mínimos necesarios en el árbol de renderizado subyacente para la transición de un estado al siguiente.

## **4.3 Insertar una imagen**

Las aplicaciones de Flutter pueden incluir tanto código como *assets*, también llamados recursos. Un asset es un archivo que se incluye e implementa con su aplicación, y es accesible en tiempo de ejecución. Los tipos comunes de recursos incluyen datos estáticos (por ejemplo, archivos JSON), archivos de configuración, iconos e imágenes (JPEG, WebP, GIF, WebP/GIF animados, PNG, BMP y WBMP).

Flutter usa el archivo **pubspec.yaml**, ubicado en la raíz de su proyecto, para identificar los recursos requeridos por una aplicación. Por ejemplo:

flutter:

assets:

- assets/icono.png

- assets/fondo.jpg

El código anterior indica que en la raíz del proyecto de Flutter tenemos una carpeta llamada “assets” en la que se encuentran los ficheros “icono.png” y “fondo.png”.

| NOTA: Es muy importante la identación del fichero **pubspec**.yaml, el espacio que debe de existir en la primera línea “flutter” respecto al origen debe de ser cero, en la segunda línea “assets” debe de tener dos espacios respecto al origen y la tercera y cuarta línea “- assets/icono.png” y “- assets/fondo.jpg” deben de tener cuatro espacios respecto al origen. |
| --- |

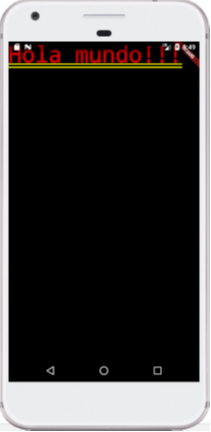
Con este código insertamos una imagen en nuestra aplicación:

| **import 'package:flutter/material.dart'**;  **void** main() {  runApp(  Image.asset(**'assets/imagen.webp'**)  );  } |
| --- |

En este código no encontramos con el widget **Image.asset** que incluye una imagen desde los assets de la aplicación.

**5.** **Estilos de texto**

**5.1 Textos con estilo único**

Avanzando un poco más en la forma de programar con Flutter, la forma normal de realizar una aplicación es de la siguiente forma:

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

**void** main() {

runApp(MyApp());

}

**class** MyApp **extends** StatelessWidget {

const MyApp({super.key});

@override

Widget build(BuildContext context) {

**return** MaterialApp(

home: Text(**'Hola mundo!!!'**),

);

}

}

**Widget y estados**

Un **widget** es una descripción inmutable de la interfaz de usuario.

Un **estado** es un cambio que se produce en un widget. Cada uno de esos cambios produce un estado en el que queda durante un tiempo determinado la aplicación. Cuando se produce un determinado evento, la aplicación produce un cambio y queda en un estado diferente a raíz del cambio producido.

## **StatefulWidget y StatelessWidget**

En el anterior programa nos encontramos con la clase Myapp que hereda de StatelessWidget:

**class** MyApp **extends** StatelessWidget {

## Nuestros widgets normalmente heredarán de uno de estos dos widgets que nos proporciona Flutter: StatefulWidget o StatelessWidget.

## La diferencia entre ambos es la persistencia de sus estados en el futuro. Cuando queremos hacer un widget que no cambie de estado internamente debemos usar StatelessWidget. Cuando decimos que no cambie su estado queremos decir que el widget sea “inmutable”.

## En caso contrario, si el widget es “mutable”, es decir, si puede cambiar dinámicamente su estado, debemos usar el StatefulWidget,

## El método build es donde creamos la interfaz de usuario para el widget ya sea un StatelessWidget o un StatefulWidget.

## **BuildContext**

## El método build es un método que hay que sobreescribir en una clase que herede de StatelessWidget y que recibe como parámetro a BuilContext. Este método debe devolver un widget.

El método build se llama cuando se inicia la aplicación o si hay algún cambio en el padre o en su hijo.

BuildContext es una referencia a la ubicación de un Widget dentro de la estructura de árbol de todos los Widgets que se construyen.

## **Widget Text**

## Un ejemplo de un widget inmutable puede ser el widget Text, en el constructor le pasamos el texto a mostrar y el Text lo guarda internamente como un atributo final. Este atributo no puede cambiar al ser el widget algo temporal y no guardar su estado.

## En el caso contrario, un widget mutable podría ser una lista de llamadas de teléfono. Cada vez que recibimos una llamada tenemos que introducirla en la lista, cambiando su contenido cada vez que esto ocurre.

Si lo único que hacemos es poner un texto en pantalla nos encontramos con el formato predefinido que se muestra en la figura, es necesario como se observa tener cierta personalización. Para ello, podemos utilizar estas instrucciones:

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

**void** main() {

runApp(MyApp());

}

**class** MyApp **extends** StatelessWidget {

@override

Widget build(BuildContext context) {

**return** MaterialApp(

home:

Center(

child:

Text(**'Este es un ejemplo de las distintas posibilidades que nos dan los textos'**,

textAlign: TextAlign.**center**,

style: TextStyle(

color: Colors.*deepOrange*,

fontSize: 25,

fontWeight: FontWeight.*w200*,

fontStyle: FontStyle.**italic**,

decoration: TextDecoration.*underline*,

decorationStyle: TextDecorationStyle.**solid**,

backgroundColor: Colors.*cyan*,

letterSpacing: 2,

height: 2,

),

),

),

);

}

}

Para centrar el texto se utiliza la siguiente instrucción:

textAlign: TextAlign.**center**,

tenemos distintas opciones de justificado dependiendo de lo que nos interese en cada momento (left, right, justify…). Esta instrucción no se encuentra en el estilo del texto.

Podemos crear un estilo personalizado añadiendo

style: TextStyle(

En este caso le hemos dado un color al texto con:

color: Colors.*deepOrange*,

un tamaño de letra:

fontSize: 25,

un ancho del carácter (más o menos negrita)

fontWeight: FontWeight.*w200*,

La hemos puesto en itálica:

fontStyle: FontStyle.**italic**,

la hemos subrayado y hemos elegido un estilo de subrayado:

decoration: TextDecoration.*underline*,

decorationStyle: TextDecorationStyle.**solid**,

hemos establecido el color cian como color de fondo del texto:

backgroundColor: Colors.*cyan*,

Le hemos establecido un espaciado entre caracteres:

letterSpacing: 2,

y, por último, se ha establecido un interlineado doble

height: 2,

Otras opciones útiles pueden ser:

a) **TextOverflow**

Esta propiedad se puede aplicar en el caso de que el texto ocupe más espacio del lugar donde se encuentre. Podemos encontrarnos con los métodos:

· **ellipsis**, añade puntos suspensivos

· **clip**, corta el contenido del texto

· **fade**, difumina el contenido

· **visible**, hace visible el texto fuera del contenedor

b) **maxLines**, que indica el número de líneas de texto que se mostrarán.

Un ejemplo puede ser:

import 'package:flutter/material.dart';

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

class TextHome extends StatelessWidget {

const TextHome({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return MaterialApp(

debugShowCheckedModeBanner: false,

home: Scaffold(

appBar: AppBar(

title: Center(child: Text("Desbordamiento del texto")),

),

body: Column(

children: [

Text("Ellipsis Example"),

Container(

padding: const EdgeInsets.all(10),

child: const Text(

"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo",

maxLines: 1,

overflow: TextOverflow.ellipsis,

),

),

Divider(),

Text("fade Example"),

Container(

width: 500,

height: 50,

padding: const EdgeInsets.all(10),

child: const Text(

"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo",

maxLines: 1,

overflow: TextOverflow.fade,

),

),

Divider(),

Text("clip Example"),

Container(

width: 500,

height: 55,

padding: const EdgeInsets.all(10),

child: const Text(

"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo",

maxLines: 1,

overflow: TextOverflow.clip,

),

),

Divider(),

Text("visible Example"),

Container(

width: 500,

height: 50,

padding: const EdgeInsets.all(10),

child: const Text(

"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo",

maxLines: 1,

overflow: TextOverflow.visible,

),

),

Divider(),

],

),

),

);

}

}

Si queremos obtener más información respecto al manejo de los textos por flutter, podemos acceder a la página:

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

**¿Cómo incluir una fuente en un proyecto?**

A menudo, la tipografía predeterminada de un proyecto no cubre todas las posibilidades que queremos dar a nuestro proyecto. Es necesario, para darle un aspecto visual mucho más atractivo incluir distintas fuentes de letra que personalice nuestra app y que dé más importancia a determinados aspectos de nuestra interfaz.

Para ello, es necesario declarar la fuente en el fichero pubspec.yaml en la sección fonts y damos la ruta de dónde encontrar nuestra fuente.

flutter:

fonts:

- family: Raleway

fonts:

- asset: fonts/Raleway-Regular.ttf

- asset: fonts/Raleway-Italic.ttf

style: italic

- family: RobotoMono

fonts:

- asset: fonts/RobotoMono-Regular.ttf

- asset: fonts/RobotoMono-Bold.ttf

weight: 700

donde:

**family**: determina el nombre de la fuente, que podemos usar en la propiedad **fontFamily** de un objeto **TextStyle**.

**asset:** es una ruta al archivo de fuente, relativa al archivo **pubspec.yaml**. Al crear nuestra aplicación, estos archivos se incluyen en el paquete de assets.

Una sola fuente puede hacer referencia a muchos archivos diferentes con diferentes pesos y estilos:

La propiedad weight especifica el peso de los contornos en el archivo como un múltiplo entero de 100 entre 100 y 900. Estos valores corresponden al FontWeight y se pueden utilizar en la propiedad fontWeight de un objeto TextStyle.

La propiedad style especifica si los contornos del archivo son italic o normal. Estos valores corresponden al FontStyle y se pueden utilizar en la propiedad fontStyle de un objeto TextStyle.

**Establecer una fuente como predeterminada**

Para usar una fuente por defecto, podemos establecer la propiedad fontFamily como parte del theme de la aplicación. El valor que proporcionamos a fontFamily debe coincidir con el family declarado en pubspec.yaml, como ya se ha indicado anteriormente.

MaterialApp(

title: 'Custom Fonts',

theme: ThemeData(fontFamily: 'Raleway'),

home: MyHomePage(),

);

**Usar la fuente en un widget específico**

Si queremos aplicar la fuente a un widget específico, como un widget Text, podemos proporcionar un TextStyle al widget.

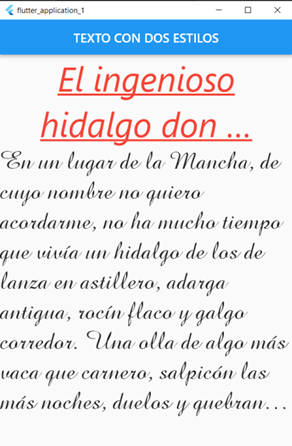
En este ejemplo, aplicaremos la fuente RobotoMono a un único widget de texto.

Text(

'Establecemos la fuente Roboto Mono sample',

style: TextStyle(fontFamily: 'RobotoMono'),

);



Si un objeto TextStyle especifica un weight o style para el que no existe un archivo de fuente exacto, el motor utiliza uno de los archivos más genéricos para la fuente e intenta extrapolar los contornos para el peso y estilo solicitados.

Un ejemplo con varios estilos de texto, podemos encontrarlo aquí:

**El fichero main.dart:**

import 'package:flutter/material.dart';

void main() => runApp(TextHome());

class TextHome extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

debugShowCheckedModeBanner: false,

home: Scaffold(

appBar: AppBar(

title: Center(child: Text("TEXTO CON DOS ESTILOS")),

),

body: Column(

children: [

Text(

"El ingenioso hidalgo don Quijote de la Mancha",

maxLines: 2,

overflow: TextOverflow.ellipsis,

textAlign: TextAlign.center,

style: TextStyle(

color: Colors.red,

fontSize: 55,

fontWeight: FontWeight.w400,

fontStyle: FontStyle.italic,

decoration: TextDecoration.underline,

),

),

Text(

"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda. El resto della concluían sayo de velarte, calzas de velludo para las fiestas, con sus pantuflos de lo mesmo, y los días de entresemana se honraba con su vellorí de lo más fino. Tenía en su casa una ama que pasaba de los cuarenta y una sobrina que no llegaba a los veinte, y un mozo de campo y plaza que así ensillaba el rocín como tomaba la podadera. Frisaba la edad de nuestro hidalgo con los cincuenta años. Era de complexión recia, seco de carnes, enjuto de rostro, gran madrugador y amigo de la caza. Quieren decir que tenía el sobrenombre de «Quijada», o «Quesada», que en esto hay alguna diferencia en los autores que deste caso escriben, aunque por conjeturas verisímiles se deja entender que se llamaba «Quijana». Pero esto importa poco a nuestro cuento: basta que en la narración dél no se salga un punto de la verdad.",

maxLines: 9,

overflow: TextOverflow.ellipsis,

fontFamily: 'Amazone-BT',

fontSize: 40,

),

),

],

),

),

);

}

}

**El fichero pubspec.yaml: (lo importante es que se incluye la fuente del texto, Amazone-BT)**

name: flutter\_application\_1

description: A new Flutter project.

# The following line prevents the package from being accidentally published to

# pub.dev using `flutter pub publish`. This is preferred for private packages.

publish\_to: 'none' # Remove this line if you wish to publish to pub.dev

# The following defines the version and build number for your application.

# A version number is three numbers separated by dots, like 1.2.43

# followed by an optional build number separated by a +.

# Both the version and the builder number may be overridden in flutter

# build by specifying --build-name and --build-number, respectively.

# In Android, build-name is used as versionName while build-number used as versionCode.

# Read more about Android versioning at https://developer.android.com/studio/publish/versioning

# In iOS, build-name is used as CFBundleShortVersionString while build-number is used as CFBundleVersion.

# Read more about iOS versioning at

# https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CoreFoundationKeys.html

# In Windows, build-name is used as the major, minor, and patch parts

# of the product and file versions while build-number is used as the build suffix.

version: 1.0.0+1

environment:

sdk: '>=2.18.2 <3.0.0'

# Dependencies specify other packages that your package needs in order to work.

# To automatically upgrade your package dependencies to the latest versions

# consider running `flutter pub upgrade --major-versions`. Alternatively,

# dependencies can be manually updated by changing the version numbers below to

# the latest version available on pub.dev. To see which dependencies have newer

# versions available, run `flutter pub outdated`.

dependencies:

flutter:

sdk: flutter

# The following adds the Cupertino Icons font to your application.

# Use with the CupertinoIcons class for iOS style icons.

cupertino\_icons: ^1.0.2

dev\_dependencies:

flutter\_test:

sdk: flutter

# The "flutter\_lints" package below contains a set of recommended lints to

# encourage good coding practices. The lint set provided by the package is

# activated in the `analysis\_options.yaml` file located at the root of your

# package. See that file for information about deactivating specific lint

# rules and activating additional ones.

flutter\_lints: ^2.0.0

# For information on the generic Dart part of this file, see the

# following page: https://dart.dev/tools/pub/pubspec

# The following section is specific to Flutter packages.

flutter:

# The following line ensures that the Material Icons font is

# included with your application, so that you can use the icons in

# the material Icons class.

uses-material-design: true

# To add assets to your application, add an assets section, like this:

# An image asset can refer to one or more resolution-specific "variants", see

# https://flutter.dev/assets-and-images/#resolution-aware

# For details regarding adding assets from package dependencies, see

# https://flutter.dev/assets-and-images/#from-packages

# To add custom fonts to your application, add a fonts section here,

# in this "flutter" section. Each entry in this list should have a

# "family" key with the font family name, and a "fonts" key with a

# list giving the asset and other descriptors for the font. For

# example:

fonts:

- family: Amazone-BT

fonts:

- asset: assets/fonts/amazone-bt.ttf

# - asset: fonts/Schyler-Italic.ttf

# style: italic

# - family: Trajan Pro

# fonts:

# - asset: fonts/TrajanPro.ttf

# - asset: fonts/TrajanPro\_Bold.ttf

# weight: 700

#

# For details regarding fonts from package dependencies,

# see https://flutter.dev/custom-fonts/#from-packages

**5.2 Textos con varios estilos**

## Para ello hace uso del widget **RichText**, que permite que un texto tenga varios estilos. En cada estilo independiente se hace uso de la clase TextSpan que representa un texto inmutable. Cada TextSpan posee la propiedad 'children' que espera recibir una lista de TextSpan, cada uno de ellos con su propio TextStyle. Si no se pone un estilo que lo sobreescriba, el TextSpan 'heredará' el estilo del TextSpan que lo contenga. Un ejemplo puede ser el siguiente:

| import 'package:flutter/material.dart';  void main() => runApp(TextosPage());  class TextosPage extends StatelessWidget {  @override  Widget build(BuildContext context) {  return MaterialApp(  home: Scaffold(  appBar: AppBar(  title: Text('Textos con diferentes estilos'),  ),  body: DefaultTextStyle(  style: TextStyle(  fontSize: 40.5,  color: Colors.blue,  fontStyle: FontStyle.italic),  child: Column(  children: <Widget>[  Text('Texto con el estilo definido en DefaultTextStyle'),  Text(  'Texto centrado en color rojo y negrilla\n Con varias líneas. Lo centra con respecto al ancho mayor de los widget´s que estén en el Column',  textAlign: TextAlign.center,  style: TextStyle(  color: Colors.red,  fontWeight: FontWeight.bold,  fontSize: 30.7),  ),  RichText(  text: TextSpan(  text: 'Este es un texto TextSpan',  style: TextStyle(  color: Colors.green,  fontSize: 25,  ),  children: [  TextSpan(  text: ' con un estilo diferente',  style: TextStyle(  color: Colors.brown,  fontSize: 25,  )),  TextSpan(  text: ' en cada parte del texto.',  style: TextStyle(  color: Colors.yellow,  fontSize: 25,  )),  TextSpan(  text:  ' Si no se pone nada, hereda el estilo del TextSpan que lo contenga'),  ]),  )  ],  ),  )));  }  } |
| --- |

## **5.3 Incluir fuentes de Google Fonts**

Para utilizar una fuente de Google Fonts es necesario utilizar el paquete google\_fonts. Un **paquete** es un código externo gratuito compartido que podemos importar en nuestra app y que nos facilitan la creación de código y nos aportan nueva funcionalidad. Podemos buscar paquetes para poderlo utilizar en nuestras aplicaciones Flutter en la página <https://pub.dev/>.

Una de las opciones muy utilizada para insertar fuentes de letras gratuitas es la de Google Fonts. Hay mucha variedad de fuentes de letras distintas. Para ello, hay que instalar el paquete de Google Fonts. En la terminal teclear:

flutter pub add google\_fonts

al realizar esto, se incorporará las dependencias necesarias para poder utilizar las fuentes de Google Fonts en el fichero pubspec.yaml:

google\_fonts: ^6.2.1

Cuando importamos algún paquete o librería, hace que el programa dependa de este código, añadiendo una **dependencia** al proyecto. Hay que añadir una entrada en la sección “dependencies” del fichero pubspec.yam. En esa dependencia se añade también la versión del código, ya que es posible que de una versión a otra cambie el constructor, los métodos…

y, posteriormente, importar el paquete correspondiente en el código. Un ejemplo puede ser el siguiente:

import 'package:flutter/material.dart';

import 'package:google\_fonts/google\_fonts.dart';

void main() {

runApp(const MyApp());

}

class MyApp extends StatelessWidget {

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

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Fuentes de letra de Google Fonts',

home: Scaffold(

appBar: AppBar(

title: const Text('Google Fonts'),

),

body: Center(

child: Text(

'Este texto es de la fuente dongle de Google Fonts',

style: GoogleFonts.fuggles(

color: Colors.indigoAccent,

letterSpacing: 8.0,

fontSize: 34,

),

))));

}

}

## **6. Trabajar con iconos**

Flutter permite introducir iconos en nuestros proyectos como ya hemos visto anteriormente. Los iconos son realmente una fuente de letra donde se encuentran todos los iconos que podemos necesitar en nuestra aplicación. Para incluir un icono tenemos que hacer uso de la clase Icon. En estas páginas tenéis una lista de los iconos disponibles:

<https://api.flutter.dev/flutter/material/Icons-class.html>

<https://material.io/resources/icons/?style=baseline>

Las opciones más interesantes son:

· size

· color

· Icons.ejemplo que elige el icono que deseemos

Un ejemplo puede ser:

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

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

**class** MyApp **extends** StatelessWidget {

@override

Widget build(BuildContext context) {

**return** MaterialApp(

title: **'ejemplo de iconos'**,

home: Scaffold(

appBar: AppBar(

title: Center(

child: Text(**'Ejemplo de inserción de iconos'**),

),

),

body:

Row(

mainAxisAlignment: MainAxisAlignment.**spaceAround**,

children: **const** <Widget>[

Icon(

Icons.*favorite*,

color: Colors.*pink*,

size: 24.0,

),

Icon(

Icons.*audiotrack*,

color: Colors.*green*,

size: 30.0,

),

Icon(

Icons.*beach\_access*,

color: Colors.*blue*,

size: 36.0,

),

],

)

)

);

}

}

**7. Disposición de elementos. Row y Column**

Cuando nos disponemos a trabajar con más de un elemento, hay dos Widgets muy simples que nos interesan en esta tarea: Row y Column. Son los que vamos a utilizar de momento para poder visualizar en nuestro móvil más de un elemento. Hay que decir que en temas posteriores se ampliarán el número de Widgets que utilizaremos para esta tarea, ya que hay gran cantidad de ellos.

**Row**: dispone la lista de Widgets en forma de filas. La forma general es

Row(

children:[

const Widget1,

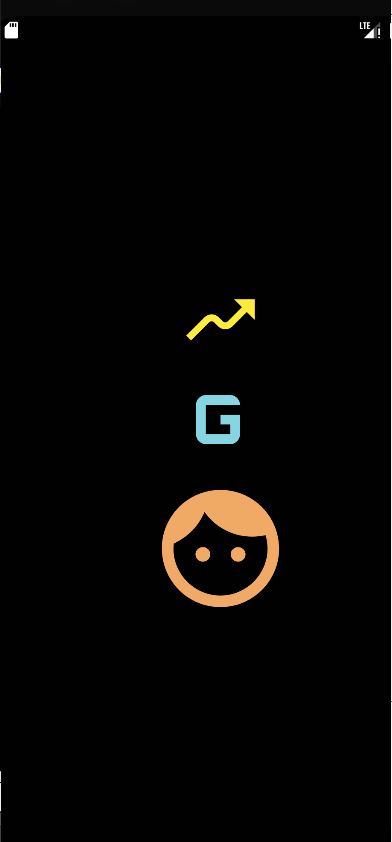
const Widget2,

const Widget3,

…

]

);

**Column**: dispone la lista de Widgets en forma de columnas y su estructura es similar a Row, aunque resultado distinto:

Colum(

children:[

const Widget1,

const Widget2,

const Widget3,

…

]

);

Podemos combinar en un documento texto, iconos e imágenes:

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

**void main(List<String> args) {**

**runApp(const Principal());**

**}**

**class Principal extends StatelessWidget {**

**const Principal({super.key});**

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**debugShowCheckedModeBanner: false,**

**home: SingleChildScrollView(**

**child: Column(children: [**

**const Center(**

**child: Text(**

**textDirection: TextDirection.ltr,**

**style: TextStyle(**

**fontSize: 40,**

**color: Colors.white,**

**decoration: TextDecoration.none),**

**"Hola mundo algo más complejo")),**

**const Icon(Icons.moving, size: 100, color: Colors.yellow**

**//textDirection: TextDirection.ltr,**

**),**

**Image.asset("assets/Images/Malaga.webp"),**

**])));**

**}**

**}**

**8****. Widget Scaffold. Drawers**

Ya lo hemos visto y utilizado, pero vamos a verlo más profundamente. Un Scafffold es el widget que implementa el aspecto visual de material design además proporciona las propiedades necesarias para mostrar un appBar, snackBars, etc. Por este motivo es el widget que contendrá nuestra aplicación.

**8.1 División del código**

Lo primero que hay que tener en cuenta cuando un programa va creciendo en el número de líneas, es dividir el código en partes funcionales para que sea más fácil de leer. Cuando nos encontramos con pocos ficheros podemos hacerlo de la siguiente forma:

1º Crea un fichero nuevo y ponle la extensión .dart. El nombre debe de iniciarse en mayúscula, por ejemplo “Pantalla.dart”.

2º Accede al fichero que va a hacer una llamada a ese código e importa el código de la forma

**import 'Pantalla.dart';**

El código del widget puede ser el siguiente:

**Main.dart**

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

**import 'Pantalla.dart';**

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

**Pantalla.dart**

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

**class MyApp extends StatelessWidget {**

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**title: 'Mi primera aplicación',**

**home: Scaffold(**

**appBar: AppBar(**

**title: Center(**

**child: Text('Barra superior'),**

**),**

**),**

**body: Center(**

**child: Text('Parte principal'),**

**),**

**floatingActionButton: FloatingActionButton(**

**child: Icon(Icons.*add*),**

**onPressed: () {},**

**),**

**),**

**);**

**}**

**}**

FloatingActionButton genera un botón en la parte inferior, que podemos personalizar:

Podemos personalizar el contenido, en vez de un icono, poner un texto de esta forma:

**Child:**

**Text(**

**"Suma",style: TextStyle(fontSize: 15),**

**),**

Podemos centrar el botón con la instrucción:

floatingActionButtonLocation: FloatingActionButtonLocation.*centerFloat*,

incluso podemos personalizar su animación con floatingActionButtonAnimator.

Podemos personalizar la parte inferior de la pantalla insertando persistentFooterButtons. Esto mostrará una fila por debajo de floatingButton.

**persistentFooterButtons: <Widget>[**

**ElevatedButton(**

**onPressed: () {},**

**child: Icon(**

**Icons.*add*,**

**color:Colors.*white*,**

**),**

**),**

**ElevatedButton(**

**onPressed: () {},**

**child: Icon(**

**Icons.*clear*,**

**color:**

**Colors.*white*,**

**),**

**),**

**],**

**8.2 Navegación entre pantallas**

La forma más simple de realizar la navegación entre pantallas, es la que vamos a explicar aquí. Más adelante en el curso se verán otras formas más sofisticadas.

Cada una de las pantallas que tenemos en nuestra aplicación se denomina Ruta. Para pasar de una ruta a otra, es necesario utilizar el widget “**Navigator**”, que es el widget encargado de gestionar las pantallas en forma de una pila.

Si queremos crear una nueva pantalla, ésta se pondrá en el primer lugar de la pila y se mostrará al usuario. Si el usuario decide volver hacia atrás o cerrar esa nueva pantalla, el Navigator la quitará de la pila dejando paso a la pantalla que estábamos viendo antes, quedará en primer lugar de la pila y por tanto se mostrará de nuevo.

En la propiedad onTap del widget Text podemos usar el widget Navigator y vamos a hacer que llame a su método push, al cual se le deben de pasar dos parámetros, el contexto actual y un **MaterialPageRoute**, que tiene un método builder que hay que implementar y que nos llevará a la nueva pantalla.

Lo que está pasando es que el Navigator crea la pantalla y la pone en la pila con el método **push**. Como está en lo más alto de la pila, esta nueva pantalla es la que ve el usuario.

Como en la pila hemos dejado debajo la pantalla inicial, **por defecto el appBar de la segunda pantalla muestra la flecha para volver atrás.** Esto también lo podemos realizar con la función pop de Navigator, que elimina el elemento actual de la cima de la pila para volver a mostrar la pantalla anterior.

**8.3 Drawer**

Es un panel que se expande y contrae desde el lado izquierdo o derecho de la pantalla del dispositivo y muestra un menú de opciones de navegación de la aplicación; es posible mostrar el menú lateral aplicando un toque en el lado izquierdo de la pantalla y arrastrando hacia la derecha o tocando el icono de la aplicación ubicado en la barra de acciones.

Si ponemos en el **main.dart** este código:

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

**import 'Screens/MenuLateral.dart';**

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

**class MyApp extends StatelessWidget {**

**const MyApp({super.key});**

**// This widget is the root of your application.**

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**title: 'Ejemplo de drawer',**

**home: Scaffold(**

**appBar: AppBar(**

**title: const Text("Ejemplo de Drawer"),**

**),**

**drawer: const MenuLateral(),**

**body: const Center(**

**child: Text("Parte principal"),**

**),**

**));**

**}**

**}**

y en el fichero MenuLateral.dart (dentro de la carpeta Screens) este otro código:

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

**import 'Enlace1.dart';**

**import 'Enlace2.dart';**

**class MenuLateral extends StatelessWidget {**

**const MenuLateral({super.key});**

**@override**

**Widget build(BuildContext context) {**

**return Drawer(**

**child: ListView(**

**children: <Widget>[**

**const UserAccountsDrawerHeader(**

**accountName: Text("Empresa"),**

**accountEmail: Text("micorreo@gmail.com"),**

**decoration: BoxDecoration(**

**image: DecorationImage(**

**image: NetworkImage(**

**"https://ichef.bbci.co.uk/news/660/cpsprodpb/6AFE/production/\_102809372\_machu.jpg"),**

**fit: BoxFit.cover)),**

**),**

**Ink(**

**color: Colors.indigo,**

**child: ListTile(**

**title: const Text(**

**"Enlace 1",**

**style: TextStyle(color: Colors.white),**

**),**

**onTap: () {**

**Navigator.of(context).pop();**

**Navigator.of(context).push(MaterialPageRoute(**

**builder: (BuildContext context) => const Enlace1()));**

**},**

**),**

**),**

**ListTile(**

**title: const Text("Enlace2"),**

**onTap: () {**

**Navigator.of(context).pop();**

**Navigator.of(context).push(MaterialPageRoute(**

**builder: (BuildContext context) => const Enlace2()));**

**},**

**),**

**],**

**),**

**);**

**}**

**}**

En el fichero Enlace1.dart (dentro de la carpeta Screens):

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

**class Enlace1 extends StatelessWidget {**

**const Enlace1({super.key});**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(**

**title: const Text("Primera pantalla"),**

**),**

**body: const Center(**

**child: Text("Contenido"),**

**),**

**);**

**}**

**}**

y en el fichero “Enlace2.dart” (dentro de la carpeta Screens):

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

**class Enlace2 extends StatelessWidget {**

**const Enlace2({super.key});**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(**

**title: const Text("Segunda pantalla"),**

**),**

**body: const Center(**

**child: Text("Parte central"),**

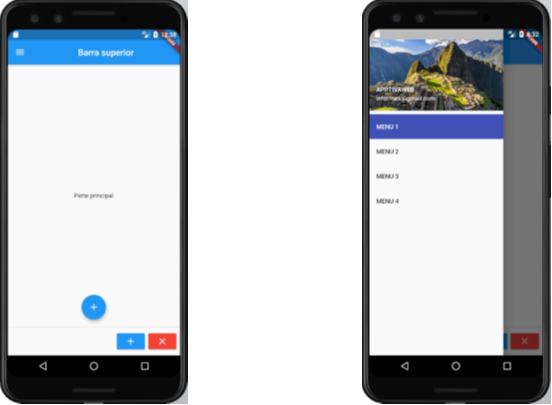
**),**

**);**

**}**

**}**

obtenemos un drawer como el que se muestra a continuación:

****

En este ejemplo hacemos uso de varios elementos que no hemos utilizado anteriormente. Podemos hacer uso de Column, pero vamos a hacer uso del widget ListView, el cual muestra una lista de elementos por pantalla. Es útil en esta situación porque permitirá a los usuarios desplazarse por el cajón si el contenido ocupa más espacio del que soporta la pantalla. Así en este ejemplo:

body: new ListView(

children: [

Text('Primer elemento!'),

Text('Segundo elemento!'),

Text('Tercer elemento!'),

]

)

Mostraría por pantalla tres líneas de texto con el texto que hemos introducido. Este widget es muy común y puede ser utilizado en distintas situaciones. En este caso lo utilizamos para mostrar los elementos del Drawer de forma ordenada. Para darle algo más de espacio entre las distintas líneas se utiliza LisTitle.

El widget UserAccountsDrawerHeader identifica al usuario de la app. Algunas de sus propiedades son las siguientes:

* accountEmail: widget que representa el email del usuario de la cuenta actual. Se muestra debajo del nombre.
* accountName: representa al nombre del usuario de la cuenta. Se muestra debajo de currentAccountPicture.
* arrowColor: representa al color de la flecha del icono
* currentAccountPicture: representa al avatar de la cuenta actual.
* decoration: establece el fondo de la cabecera. Si no está definido se usará el color por defecto.
* margin: margen alrededor de la cabecera.

**Hay otras propiedades interesantes en Scaffold, como pueden ser:**

## **backgroundColor:** establece el color que se desea tenga en el fondo del Scaffold es un tipo [**Color**](https://docs.flutter.io/flutter/dart-ui/Color-class.html).

## **endDrawer**: es un drawer que se desliza de derecha a izquierda o de izquierda a derecha según el uso puede ser según se necesite.

## **bottomSheet**: una hoja para mostrar contenido que aparece desde la parte inferior del Scaffold. Esta se puede usar para mostrar si se desea un menú, el contenido o algún mensaje informativo.

**snackBar:** barra inferior que aparece durante unos segundos con información.

**9****. Finalización de una apk con flutter**

Hay varios pasos que tenemos que realizar para poder generar una apk de nuestra aplicación sin problemas. Para ello, podemos visitar el siguiente enlace de la documentación oficial donde nos explican todos los detalles a tener en cuenta:

<https://docs.flutter.dev/deployment/android>