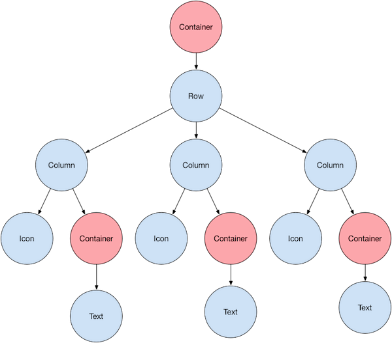
TRABAJO DAM: DESARROLLO CON FLUTTER

1. Objetivos

El objetivo de este trabajo es aprender el manejo básico del SDK de Flutter y hacer destacar las principales diferencias con otros lenguajes/frameworks que hemos aprendido en la asignatura. Se va a crear, compilar y ejecutar una aplicación con Flutter que valdrá como calculadora básica.

1. Introducción y breves apuntes teóricos

Flutter es un framework para el desarrollo de aplicaciones multiplataforma y soporte web. Utiliza Dart como lenguaje de programación, ambos desarrollados por Google. Los principales puntos a destacar de Flutter son la rápida capacidad de desarrollo, puesto que pueden verse los cambios a tiempo real, su versatilidad a la hora de plantear las interfaces y el hecho de que crea una app nativa para cada sistema operativo sin modificar el código. La potencia de Dart y la sencillez de codificación, junto con su rápido renderizado la hacen una opción a tener en cuenta a la hora de desarrollar aplicaciones para varias plataformas.

El paradigma de codificación de simple pero muy potente: todo es un widget. Durante el desarrollo, veremos cómo todo es un widget que puede incorporar widgets dentro. Esto hace que las interfaces gráficas tengan una apariencia moderna y estructurada sin mucha dificultad de codificación.

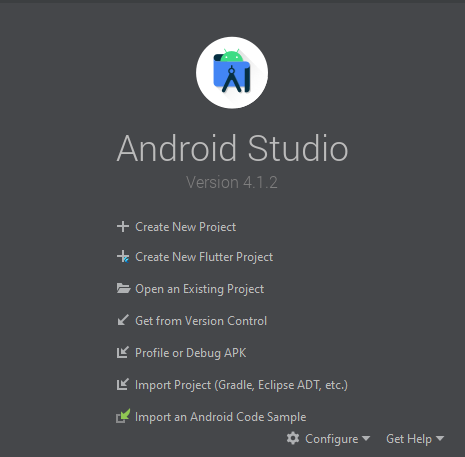
1. Requisitos para la práctica

• Acceso a internet para consulta de documentación

• Android Studio

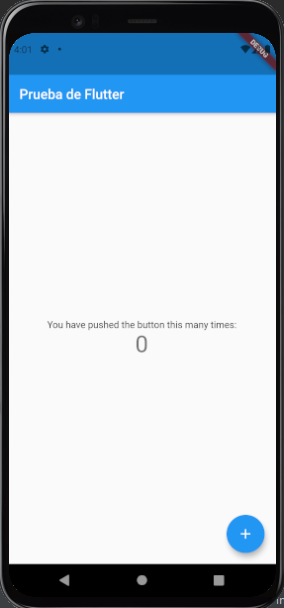
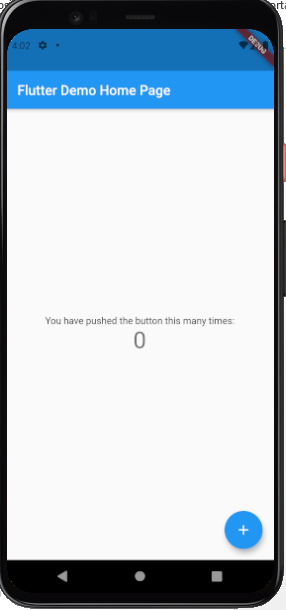
• Flutter

1. Desarrollo de la práctica
2. Introducción

En la ventana principal de Android Studio, una vez instalado Flutter, nos aparecerá lo siguiente:

Pulsaremos en Create New Flutter Project->Flutter Application y rellenaremos los campos de forma pertinente.

En un inicio lo que encontraremos será un esqueleto de código predeterminado que nos muestra una aplicación con un pulsador. Para probar la capacidad de Flutter de desarrollo en tiempo real o “Hot Reload”, pruebe a modificar el home de la primera clase y cambiar el texto. Verá que sólo pulsando en el botón de sincronización, sin reiniciar la app, los cambios se aplican.



Si ojea el código predeterminado, ahí se explican brevemente los fundamentos de cómo está orientado el código.

En una primera instancia, está la clase MyApp que extiende a StatelessWidget. Estos widget sin estado se suelen utilizar para situaciones en las que ningún uso que se le dé a la aplicación afecta a los parámetros configurados en este. Incorporar un reloj por ejemplo en un StatelessWidget sería no recomendable. Como hemos comentado previamente, todo en Flutter son widgets. Éste será el raíz.

A continuación vemos la clase MyHomePageState, que extiende a StatefulWidget. Como se puede imaginar, un widget con estado permite cambios en las configuraciones de éste durante su ciclo de vida. Las instancias de StatefulWidget son inmutables en sí mismas, su estado se guarda a parte en métodos creados por createState o en objetos a los que se suscribe.

Por último, vemos otra clase llamada \_MyHomePageState que extiende a la clase State. Aquí se definirá la lógica y el estado interno del widget anterior.

Una vez que hemos sobrevolado las clases que tiene predeterminadas Flutter cuando creamos un proyecto, vamos a hacer unos cambios minoritarios para terminar de explicar cómo se orientan los layouts en Flutter.

Vamos a añadir un reloj que se actualiza en tiempo real. Para ello, lo primero que debemos hacer es ejecutar en un terminal, en la carpeta del proyecto:

*flutter pub add intl*

Con esto nos aseguramos de incluir una librería necesaria para la modificación.

NOTA: Si no se reconociera el comando flutter, debemos añadir como variable de entorno a la consola la carpeta ~/flutter/bin.

Añadiremos las siguientes dependencias al programa:

|  |
| --- |
| import 'dart:async'; import 'package:flutter/material.dart'; import 'package:intl/intl.dart'; |

Añada estos dos métodos a \_MyHomePageState y cree la variable de instancia

*String \_timeString*

|  |
| --- |
| void \_getTime() {  final String formattedDateTime =  DateFormat('yyyy-MM-dd \n kk:mm:ss').format(DateTime.now()).toString();  setState(() {  \_timeString = formattedDateTime;  }); } @override void initState() {  super.initState();  Timer.periodic(Duration(seconds: 1), (Timer t) => \_getTime()); } |

Observe cómo en el método de initState hemos creado un temporizador que dura 1 segundo que irá actualizando sucesivamente el widget.

Luego, añadiremos un hijo más a Column, que nos mostrará la hora en el formato definido anteriormente:

|  |
| --- |
| child: Column(  mainAxisAlignment: MainAxisAlignment.center,  children: <Widget>[  Text(  'You have pushed the button this many times:',  ),  Text(  '$\_counter',  style: Theme.*of*(context).textTheme.headline4,  ),  Text('$\_timeString',  style: Theme.*of*(context).textTheme.headline5,),  ],  ) ), |

Si sincroniza el simulador podrá ver como bajo el texto ahora aparece un reloj que da la fecha y hora y se actualiza cada segundo. Este pequeño añadido nos permite conocer un poco mejor la estructura del desarrollo de widgets en Flutter:

Como hemos explicado anteriormente, los widgets son subclases de o bien StatefulWidget o bien de StatelessWidget, en función de cómo trabajan los estados. Dentro de cualquiera, debe existir la función build que describe sus subniveles. Los widgets básicos son Text, Row, Column, Stack y Container. Por explicarlo superficialmente, Stack y Container son dos subtipos que permiten más movilidad a sus widgets hijos.

A continuación vamos a borrar todo el código para crear una calculadora.

1. Interfaz gráfica de la calculadora

Lo primero que haremos será borrar todo el código del main e introducir el siguiente esqueleto del que partiremos:

|  |
| --- |
| import 'package:flutter/material.dart';  void main() {  runApp(Calculadora()); }  class Calculadora extends StatelessWidget {    @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Practica Calculadora',  theme: ThemeData(  primarySwatch: Colors.*blueGrey*,  ),  home: MyHomePage(title: 'Calculadora en Flutter'),  );  } }  class MyHomePage extends StatefulWidget {  MyHomePage({Key key, this.title}) : super(key: key);  final String title;   @override  \_MyHomePageState createState() => \_MyHomePageState(); }  class \_MyHomePageState extends State<MyHomePage> {  @override  void initState() {  }  @override  Widget build(BuildContext context) {  return Scaffold(  );  } } |

Con el siguiente esqueleto estaremos trabajando sólo con lo imprescindible. Es importante destacar que el método build devuelve un Scaffold o *andamio* que es una clase básica que implementa los widgets que hemos comentado anteriormente y facilita el diseño.

Empezaremos desarrollando la interfaz gráfica de nuestro proyecto. Para ello, añadiremos lo siguiente:

Añada la siguiente clase a main.dart:

|  |
| --- |
| class Boton extends StatelessWidget {  final String simbolo;  final int colorFondo;  final double ancho;  final double largo;  final int colorTxt;  final Function funcion;   const Boton({  Key key,  this.simbolo,  this.ancho=65,  this.largo=65,  this.colorFondo=0xFF000000,  this.colorTxt=0xFF69F0AE,  this.funcion,  }) :super(key: key);   @override  Widget build(BuildContext context) {  return Container(  margin: EdgeInsets.all(1),  color: Colors.*greenAccent*,  child: SizedBox(  width: ancho,  height: largo,  child: Card (  color: Color(colorFondo),  child: InkWell(  onTap:(){  funcion(simbolo);  },  child: Center(  child:Text(  simbolo,  style: TextStyle(  color: Color(colorTxt),  fontSize: 20),  textAlign: TextAlign.center  )  )  )  )  )  );  } } |

Esta clase nos servirá para establecer un formato de botón con unos colores de texto, borde y fondo predeterminados. Alteraremos algunos de ellos al crear los botones en el widget principal para diferenciar los números de los símbolos.

A continuación, edite la clase MyHomePageState con el siguiente código:

|  |
| --- |
| class \_MyHomePageState extends State<MyHomePage> {  @override  void initState() {  }   @override  Widget build(BuildContext context) {  return Scaffold(   body: Container(  padding: EdgeInsets.all(12),  color: Colors.*black*,  child: Column(  mainAxisAlignment: MainAxisAlignment.center,  children: [  Container(  color: Colors.*black*,  height: 50,  width: 300,  alignment: Alignment.*centerLeft*,  child: FittedBox(  alignment: Alignment.*topLeft*,  child: Text(  \_ans,  style: TextStyle(color: Colors.*greenAccent*, fontSize: 30),  )  )  ),  Container(  color: Colors.*greenAccent*,  height: 100,  width: 300,  alignment: Alignment.*centerRight*,  child: FittedBox(  alignment: Alignment.*topRight*,  child: Text(  \_expresion,  style: TextStyle(color: Colors.*black*, fontSize:40),  ),  )  ),  Container(  color: Colors.*black*,  height: 100,  width: 300,  alignment: Alignment.*centerRight*,  child: FittedBox(  alignment: Alignment.*topRight*,  )  ),  Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Boton(  simbolo: "AC", colorFondo :0xDD000000, funcion:allClear  ),  Boton(  simbolo: "C", colorFondo :0xDD000000, funcion:clear  ),  Boton(  simbolo: "(", colorFondo :0xDD000000, funcion:click  ),  Boton(  simbolo: ")", colorFondo :0xDD000000, funcion:click  )  ],  ),  Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Boton(  simbolo: '1', funcion:click  ),  Boton(  simbolo: '2', funcion:click  ),  Boton(  simbolo: "3", funcion:click  ),  Boton(  simbolo: "+", colorFondo :0xDD000000, funcion:click  )  ],  ),  Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Boton(  simbolo: '4',funcion:click  ),  Boton(  simbolo: '5',funcion:click  ),  Boton(  simbolo: "6",funcion:click  ),  Boton(  simbolo: "-", colorFondo :0xDD000000, funcion:click  )  ],  ),  Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Boton(  simbolo: '7', funcion:click  ),  Boton(  simbolo: '8', funcion:click  ),  Boton(  simbolo: "9", funcion:click  ),  Boton(  simbolo: "/", colorFondo :0xDD000000, funcion:click  )  ],  ),  Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Boton(  simbolo:"=",  funcion:solucionar,  colorFondo :0xAA000000,  ancho: 153  ),  Boton(  simbolo:"0",  funcion:click,  ),  Boton(  simbolo:"\*",  funcion:click,  colorFondo :0xDD000000,  ),  ]  )  ],  )  )  );  } } |

Fíjese en los dos container hijos de la columna principal. Ambos son una caja a través de la cual imprimiremos el texto introducido por teclado y en otro la solución. La reubicación depende de cómo implementemos la lógica en el apartado siguiente.

El aspecto que debería tener el terminal si presiona ahora el botón de HotReload (el rayo) es este. Como puede ver, hemos creado un Layout a base de crear widgets que incorporan uno o varios hijos. Es importante destacar que hemos incorporado un widget vacío del color del fondo para separar el teclado de la solución y el historial, ya que Flutter entiende el uso de los widgets también a modo de separadores.

Cuando acabemos con la lógica de la aplicación, cambiaremos el estilo de las ventanas una vez más. Fíjese en que los números y el resto de botones tienen un color de fondo diferente y repare en que esto se debe a que existe un colorFondo en la clase botón que puede ser sobreescrito cuando se implementa el widget y se le pasa otro valor a ese atributo.

1. Lógica de la calculadora

Una vez hemos estructurado el layout en gran parte, tenemos que dotar a los widgets de capacidad de operación si es necesario. Como hemos comentado, algunos widgets se implementan vacíos para poder dimensionar el layout a nuestro gusto. Nos centraremos en los widgets con capacidad de operación.

Para esta tarea tendremos que desarrollar varias funciones que nos permitan evaluar la expresión matemática, ponerla en tiempo real en el recuadro y guardar la última solución en el histórico. Empezaremos añadiendo la librería que usaremos para la evaluación de expresiones:

|  |
| --- |
| import 'package:math\_expressions/math\_expressions.dart'; |

NOTA: si el editor se lo marca como error significa que no tiene la librería instalada. Para instalarla, sitúese en el directorio del proyecto con un terminal y ejecute

*flutter pub add math\_expressions*

A continuación añadiremos las funciones que se encargarán de manipular los datos. En la clase \_MyHomePageState añada las siguientes variables globales y estas funciones.

|  |
| --- |
| String \_expresion=''; String \_ans=''; void click(String texto){  setState(()=> \_expresion += texto); } void clear(String texto){  setState((){  \_expresion='';  }); } void allClear(String texto){  setState((){  \_expresion='';  \_ans='';  }); } void solucionar(String expr){  Parser p = Parser();  Expression exp = p.parse(\_expresion);  ContextModel cm = ContextModel();  setState(() {  \_ans = exp.evaluate(EvaluationType.REAL, cm).toString();  \_expresion = '';  }); } |

Es importante destacar que estas funciones serán asociadas al atributo función que tienen los Botones. Cada botón implementará una función en concreto según su naturaleza. Así, los números y operaciones usarán la función click, puesto que queremos que se escriban en la pantalla. El símbolo de igual ejecutará Solucionar, que con ayuda de la librería evaluará la expresión y la resolverá. Clear y AllClear responden al comportamiento normal de los botones C y AC de una calculadora básica.

En un ejemplo podemos ver en la captura de la izquierda la introducción de una expresión cualquiera. Al pulsar el botón igual (ver captura 2) la solución aparecerá en el histórico y tendremos la ventana libre para escribir otra expresión nueva. Al pulsar el botón igual se repetirá el proceso.





1. Exportando a Android

En Android Studio, construya la APK y arrástrela a un terminal. Se instalará y ejecutará como una aplicación Android nativa.

