**ORGANIZACIÓN PROYECTO DE FLUTTER**

Para organizar los archivos en un proyecto de Flutter, es importante seguir una estructura clara y lógica que facilite la gestión y el mantenimiento del proyecto. Aunque Flutter no proporciona una recomendación específica para la estructuración de la aplicación, se pueden seguir ciertos enfoques para lograr una estructura de archivos óptima. A continuación, se presenta un ejemplo de cómo organizar los archivos en un proyecto de Flutter:

**1. \*\*Organización básica de archivos y carpetas:\*\***

- `lib`: Esta es la carpeta principal donde se encuentra todo el código fuente de la aplicación.

- `main.dart`: Archivo principal que contiene el punto de entrada de la aplicación.

- `app.dart`: Archivo que contiene la configuración general de la aplicación, como temas y rutas.

- `screens`: Carpeta que contiene todos los archivos de pantalla de la aplicación.

- `home.dart`: Archivo de pantalla de inicio.

- `profile.dart`: Archivo de pantalla de perfil.

- `widgets`: Carpeta que contiene todos los archivos de widgets personalizados.

- `custom\_button.dart`: Archivo de widget de botón personalizado.

- `models`: Carpeta que contiene todos los archivos de modelos de datos.

- `user.dart`: Archivo de modelo de usuario.

- `services`: Carpeta que contiene todos los archivos de servicios, como API y bases de datos.

- `api.dart`: Archivo de servicio de API.

- `utils`: Carpeta que contiene archivos de utilidades y funciones comunes.

- `constants.dart`: Archivo que contiene constantes de la aplicación, como colores y estilos.

- `assets`: Carpeta que contiene todos los recursos de la aplicación, como imágenes y fuentes.

- `images`: Carpeta que contiene todas las imágenes de la aplicación.

- `fonts`: Carpeta que contiene todas las fuentes de la aplicación.

[Source 0](https://barcelonageeks.com/flutter-estructura-de-archivo/)

**2. \*\*Uso de `pubspec.yaml` para gestionar las dependencias y recursos:\*\***

El archivo `pubspec.yaml` es el archivo de configuración del proyecto que se utilizará mucho al trabajar con el proyecto Flutter. Este archivo contiene la configuración general del proyecto, como el nombre, la descripción y la versión del proyecto, así como los recursos (por ejemplo, imágenes) que deben estar disponibles en la aplicación Flutter.

[Source 2](https://ichi.pro/es/comenzando-con-flutter-2-estructura-del-proyecto-141178760614512)

**3. \*\*Uso de archivos y carpetas específicos de Git y Flutter:\*\***

- `.gitignore`: Este archivo contiene una lista de archivos, extensiones de archivo y carpetas que deben ignorarse al trabajar con Git.

- `.metadata`: Este archivo es administrado por Flutter automáticamente y se utiliza para rastrear las propiedades del proyecto Flutter.

- `.packages`: Este archivo contiene contenido generado automáticamente por el SDK de Flutter y se utiliza para contener una lista de dependencias para el proyecto de Flutter.

- `<project\_name>.iml`: Este archivo contiene más configuraciones del proyecto Flutter y generalmente no se modifica manualmente.

[Source 2](https://ichi.pro/es/comenzando-con-flutter-2-estructura-del-proyecto-141178760614512)

En resumen, la estructura de archivos en un proyecto de Flutter debe ser clara y lógica para facilitar la gestión y el mantenimiento del proyecto. La organización básica de archivos y carpetas, el uso de `pubspec.yaml` y los archivos y carpetas específicos de Git y Flutter son aspectos importantes a tener en cuenta al organizar un proyecto de Flutter.

**DOCUMENTACIÓN APP FLUTTER DADOS**

1. Creamos una aplicación en VS Code de flutter llamada flutter\_dados\_1.
2. Borramos todo el contenido de main.dart y empezamos desde cero.
3. Lo primero que debemos es importar la librería material.dart con la siguiente instrucción: import ‘package:flutter/material.dart’
4. Luego agregamos la función principal:

//1. función principal para correr la aplicación

void main() {

  runApp(MaterialApp(home: Text('Hola Mundo')));

}

Con lo anterior, podemos ejecutar nuestra maquina virtual y obtener el siguiente resultado, que lo que hemos dicho en el código es llamar una función que no tiene retorno llamada main() y que dentro de ella se dice corra la aplicación llamada “MatarialApp”, que a su vez, indica que en el cuerpo de la aplicación aparezca el texto “Hola Mundo”.



Lo anterior no es una aplicación bonita, pero Flutter ya nos ha creado la primera interfaz de usuario.

Para hacerlo mas bonito tenemos que agregar mas Widgets, solamente tenemos una combinación muy básica de Widgets.

Para ello vamos a utilizar el Widget Scaffold (Andamio), Este widget está diseñado para ser la raíz de una jerarquía de widgets y puede tener una barra de aplicaciones, un cuerpo y un Drawer (menú lateral) entre otros elementos.

El Scaffold se utiliza para implementar la estructura básica de una aplicación y proporciona una serie de elementos preconstruidos que siguen las directrices de Material Design.

Estas son las modificaciones realizadas al código anterior:

void main() {

  runApp(

    const MaterialApp(

      home: Scaffold(

        body: Center(

          child: Text('Hola Mundo'),

        ),

      ),

    ),

  );

}

Y esta es la apariencia de nuestra App:

Imagen que contiene Gráfico

Descripción generada automáticamente

1. Cambiando la presentación de la interfaz:

VS Code, permite envolver en un widget otro widget, esto permite mejorar las características, en este caso, vamos a incorporar un widget container, que permitirá mejorar las características de presentación de la interfaz.

Con el container, vamos a agregar un color a la interfaz, para ello vamos a trabajar con un color gradiente.

void main() {

  runApp(

    MaterialApp(

      home: Scaffold(

        //Agregamos un container con un gradiente

        body: Container(

          decoration: const BoxDecoration(

            gradient: LinearGradient(

              colors: [

                Colors.blue,

                Colors.green,

              ],

              //agregar inicio y final del gradiente

              begin: Alignment.topLeft,

              end: Alignment.bottomRight,

            ),

          ),

          child: const Center(

            child: Text('Hola Mundo'),

          ),

        ),

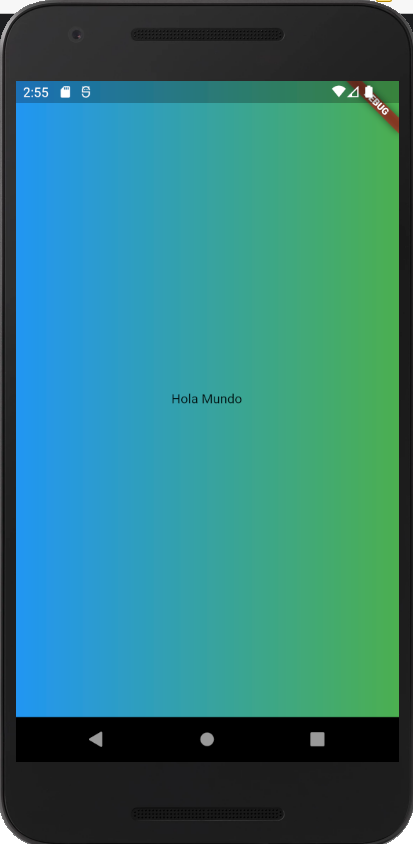
      ),

    ),

  );

}

Este es el resultado:



1. Colocando estilo al texto.

Para agregar un estilo al texto que se ve en la interfaz del usuario, podemos agregar los siguientes estilos, dentro de la palabra TextStyle, así:

            child: Text(

              'Hola Mundo',

              //Agregamos estilo al texto

              style: TextStyle(

                  fontSize: 30,

                  color: Colors.white,

                  fontWeight: FontWeight.bold,

                  fontStyle: FontStyle.italic),

            ),

Con ello, la interfaz del usuario quedará de la siguiente manera:

Al texto le agregamos un tamaño de letra de 30 px, un color blanco con negrita y un estilo italic.

Para ver las características que se pueden aplicar a un Widget, se presiona las teclas Ctrl + Space y con ello, se puede ver todas las alternativas a utilizar.

**CREACIÓN DE WIDGETS PERSONALIZADOS**

Cuando el código o árbol de un archivo de flutter se hace cada vez masa grande, la recomendación es dividir ese código en partes más pequeñas, y luego a través de un llamado, acoplar esas partes del código para en conjunto hagan para lo que fueron creadas…

Para poder separar o dividir un árbol, lo primero que debemos hacer es crear una clase utilizando la palabra clave de Dart **Class**, las clases permiten definir el tipo de datos que se almacenarán en un objeto cuando se cree en tiempo de ejecución y define que funciones adicionales, denominadas métodos, pueden almacenarse en dicho objeto.

Escribimos class seguida del nombre que queramos darle a la clase, los nombres de las clases deben comenzar con una letra mayúscula, si el nombre se compone por varias palabras, no deben estar separadas por espacios o guiones, y cada letra inicial de la palabra deberá también comenzar con mayúsculas; el nombre debe estar relacionado con lo que va a contener el widget que se está creando.

Posteriormente al nombre, como es una clase que heredará de otra clase algunas características, debe tener la palabra clave “extends” y luego el nombre de la clase de la que se quiere heredar.

Para crear una clase en VScode, escribimos las iniciales STF y seleccionamos el tipo de widget que heredara, si es con estado (statefulWidget) o sin estado (StatelessWidget), y agregamos el nombre de la clase.

class GradientContainer extends StatelessWidget {

  const GradientContainer({super.key});

  @override

  Widget build(BuildContext context) {

    return const Placeholder();

  }

}

Ahora bien, con esta clase lo que queremos es que nos devuelva otro widget y que esto es parte del código que deseamos dividir, entonces, esa parte del código grande la cortamos y la colocamos en esta clase que estamos creando.

Este es el estado inicial del código de main()

import 'package:flutter/material.dart';

void main() {

  runApp(

    MaterialApp(

      home: Scaffold(

        //Agregamos un container con un gradiente

        body: Container(

          decoration: const BoxDecoration(

            gradient: LinearGradient(

              colors: [

                Colors.blue,

                Colors.green,

              ],

              //agregar inicio y final del gradiente

              begin: Alignment.topLeft,

              end: Alignment.bottomRight,

            ),

          ),

          child: const Center(

            //Agregamos un texto con estilo

            child: Text(

              'Hola Mundo',

              //Agregamos estilo al texto

              style: TextStyle(

                fontSize: 30,

                color: Colors.white,

              ),

            ),

          ),

        ),

      ),

    ),

  );

}

class GradientContainer extends StatelessWidget {

  const GradientContainer({super.key});

  @override

  Widget build(BuildContext context) {

    return const Placeholder();

  }

}

Y ahora este es el resultado final

import 'package:flutter/material.dart';

void main() {

  runApp(

    const MaterialApp(

      home: Scaffold(

        //Agregamos un container con un gradiente

        body: GradientContainer(),

      ),

    ),

  );

}

//Creación de un widget personalizado

class GradientContainer extends StatelessWidget {

  const GradientContainer({super.key});

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: const BoxDecoration(

        gradient: LinearGradient(

          colors: [

            Colors.blue,

            Colors.green,

          ],

          //agregar inicio y final del gradiente

          begin: Alignment.topLeft,

          end: Alignment.bottomRight,

        ),

      ),

      child: const Center(

        //Agregamos un texto con estilo

        child: Text(

          'Hola Mundo',

          //Agregamos estilo al texto

          style: TextStyle(

            fontSize: 30,

            color: Colors.white,

          ),

        ),

      ),

    );

  }

}

Entonces, el contenedor lo trasladamos a la nueva clase y en su lugar, colocamos el nombre de la nueva clase acompañada de paréntesis para que la llame y realice lo que está dentro de ella.

Al ejecutar el programa, la aplicación no ha cambiado, sigue siendo la misma, pero internamente, hemos dado una mejor organización a nuestro código.

**DIVIDIR CÓDIGO ENTRE ARCHIVOS**

1. Crear un nuevo archivo con código de otro archivo.

Las nuevas clases personalizadas pueden ser cortadas y crear un nuevo archivo .dart con el nombre de la clase, pero en minúsculas; para ello, del código anterior, cortamos el código de la clase GradientContainer(), vamos a la carpeta lib y dentro de ella, creamos el archivo llamado “gradient\_container.dart” que como se puede dar cuenta, el nombre del archivo si admite guiones entre palabras, lo que no admite son las mayúsculas.

Y al código restante, como queda la clase GradientContainer() con error, para solucionarlo, debemos importar el archivo que acabamos de crear, así queda el código del archivo main().

import 'package:flutter/material.dart';

import 'gradient\_container.dart';

void main() {

  runApp(

    const MaterialApp(

      home: Scaffold(

        //Agregamos un container con un gradiente

        body: GradientContainer(),

      ),

    ),

  );

}

Al correr nuevamente la aplicación, el programa sigue funcionando sin ningún problema.

**VARIABLES**

Las variables permiten hacer más potente nuestra aplicación y construirlas hará que al querer mejorar o cambiar algún dato, solo debamos realizarlo en la variable y no buscarla en todo el código.

Las variables pueden estar dentro de una clase o fuera de ella, para poner en práctica la creación de variables, trabajaremos el archivo gradeint\_container.dart.

El inicio y final del gradient lo vamos a convertir en variables, entonces, este es el código inicial sin creación de variables:

import 'package:flutter/material.dart';

import 'package:flutter\_dado\_1/style\_text.dart';

//Creación de un widget personalizado

class GradientContainer extends StatelessWidget {

  const GradientContainer({super.key});

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: const BoxDecoration(

        gradient: LinearGradient(

          colors: [

            Colors.blue,

            Colors.green,

          ],

          //agregar inicio y final del gradiente

          begin: Alignment.topLeft,

          end: Alignment.bottomRight,

        ),

      ),

      child: const Center(

        //Agregamos un texto con estilo

        child: StyleText(),

      ),

    );

  }

}

Este es nuestro código final con variables y las modificaciones que sufre el código anterior.

//Creacion de variable para el inicio del gradiente

var startAlignment = Alignment.topLeft;

//Creacion de variable para el final del gradiente

var endAlignment = Alignment.bottomRight;

//Creación de un widget personalizado

class GradientContainer extends StatelessWidget {

  const GradientContainer({super.key});

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          colors: const [

            Colors.blue,

            Colors.green,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: const Center(

        //Agregamos un texto con estilo

        child: StyleText(),

      ),

    );

  }

1. Utilización de variables para optimizar la clase StyleText.

Dentro de la clase StyleText, el texto es intocable o esta predefinido, vamos a crear variables que permitan que el texto sea modificado de manera automática de acuerdo a las funciones que se desarrollen, este es el código inicial:

class StyleText extends StatelessWidget {

  const StyleText({super.key});

  @override

  Widget build(BuildContext context) {

    return const Text(

      'Hola Mundo',

      //Agregamos estilo al texto

      style: TextStyle(

        fontSize: 30,

        color: Colors.white,

      ),

    );

  }

}

Ahora, este será el código final optimizado y reutilizable.

class StyleText extends StatelessWidget {

  const StyleText(this.text, {super.key});

  final String text;

  @override

  Widget build(BuildContext context) {

    return Text(

      text,

      //Agregamos estilo al texto

      style: const TextStyle(

        fontSize: 30,

        color: Colors.white,

      ),

    );

  }

}

Y cuando se llama a la clase, debe estar acompañado del texto a mostrar, ejemplo:

child: const Center(

        //Agregamos un texto con estilo

        child: StyleText('Hola Mundo!'),

Lo mismo se hará con la clase GradientContainer, para hacer más optimo el código, especialmente con los colores.

//Creación de un widget personalizado

class GradientContainer extends StatelessWidget {

  const GradientContainer(this.colorOne, this.colorTwo, {super.key});

  //Creamos las variables para los colores

  final Color colorOne;

  final Color colorTwo;

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          //Agregamos las valiables colores

          colors: [

            colorOne,

            colorTwo,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: const Center(

        //Agregamos un texto con estilo

        child: StyleText('Hola Mundo!'),

      ),

    );

  }

}

El llamado de la función por parte de main() sería:

body: GradientContainer(

            Color.fromARGB(255, 3, 58, 102), Color.fromARGB(255, 73, 49, 129)),

**Visualización de imágenes**

1. Agregar imágenes al proyecto de flutter.

Lo primero que se debe hacer es crear un directorio dentro de la raíz del proyecto llamado “assets” esta carpeta almacenará todos los archivos de imágenes, videos y demás archivos multimedia que se utilizarán en el proyecto.

Dentro de la carpeta creada, creamos un nuevo directorio llamado “images” y dentro de ella, copiaremos las imágenes a utilizar en el proyecto.

Ahora, para incorporar estas imágenes dentro del proyecto, debemos ir al archivo pubspec.yaml y agregar las siguientes líneas para que estas imágenes sean aceptadas por el proyecto, en caso contrario no se podrán vincular.

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

  assets:

    - assets/images/dice-1.png

    - assets/images/dice-2.png

    - assets/images/dice-3.png

    - assets/images/dice-4.png

    - assets/images/dice-5.png

    - assets/images/dice-6.png

Ahora, para probar que funcionan, vamos al gradientcontainer y vamos a reemplazar el styletext por una función que nos permita agregar una imagen.

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          //Agregamos las valiables colores

          colors: [

            colorOne,

            colorTwo,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: Center(

        //Agregamos un texto con estilo

        //child: StyleText('Hola Mundo!'),

        //Agregaremos ahora una imagen

        child: Image.asset(

          'assets/images/dice-1.png',

          width: 200,

        ),

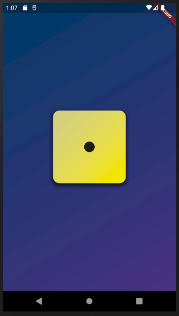
      ),

    );

  }

}

1. Agregar botones y funciones como valores.

Hasta el momento tenemos la siguiente interfaz de usuario con una imagen de un lado del dado, ahora queremos agregar a esta interfaz un botón que permita interactuar con el dado y muestre otras caras de este de manera aleatoria.

Para ello, debemos dentro del widget Center, crear otro widget “Columna” que permitirá crear un widget encima de otro widget o en otras palabras, organizar de manera vertical nuestros widgets, si queremos en forma horizontal, entonces hablamos de “filas”.

Los cambios hechos en el archivo gradient\_container.dart, asi queda:

 //Creamos una funcion para generar un numero aleatorio y tirar los dados

  void tirarDados() {

    //Creamos una variable para guardar el numero aleatorio

    final numeroAleatorio = 1 + Random().nextInt(6);

    //Imprimimos el numero aleatorio

    print(numeroAleatorio);

  }

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          //Agregamos las valiables colores

          colors: [

            colorOne,

            colorTwo,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: Center(

        //Agregamos un texto con estilo

        //child: StyleText('Hola Mundo!'),

        //Agregaremos ahora una imagen

        child: Column(

          //Como quieremos que la imagen y el texto esten centrados, los agrupamos en una lista dentro del widget Column

          children: [

            Image.asset(

              'assets/images/dice-1.png',

              width: 200,

            ),

            TextButton(

              //En on pressed agregamos una funcion que la creamos despues de las variables

              onPressed: tirarDados,

              child: const Text('Tirar Dado'),

            ),

          ],

        ),

      ),

    );

  }

}

1. Botones de estilo y trabajando con relleno.

Al aplicar los anteriores cambios en el código, la interfaz gráfica sufrió un cambio inesperado y se ubico en la parte superior a pesar de que el contenido del mismo esta dentro del widget “Center” y esto se debe a que como envolvimos tanto la imagen como el botón texto dentro de una columna, esto hacer que la columna ocupe todo el espacio vertical de la pantalla; entonces, vamos a agregar unos márgenes para que la interfaz vuelva a su estado inicial.

Esto fue lo que agregamos:

          //Agregamos el main axis alignment para centrar el contenido

          mainAxisSize: MainAxisSize.min,

Ahora, procedemos a darle un estilo al texto del botón “TextButton”:

@override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          //Agregamos las valiables colores

          colors: [

            colorOne,

            colorTwo,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: Center(

        //Agregamos un texto con estilo

        //child: StyleText('Hola Mundo!'),

        //Agregaremos ahora una imagen

        child: Column(

          //Agregamos el main axis alignment para centrar el contenido

          mainAxisSize: MainAxisSize.min,

          //Como quieremos que la imagen y el texto esten centrados, los agrupamos en una lista dentro del widget Column

          children: [

            Image.asset(

              'assets/images/dice-1.png',

              width: 200,

            ),

            //Agregaremos una caja que ocupara un espacio dentro de la interfaz pero que no tendra contenido

            const SizedBox(

              height: 20,

            ),

            TextButton(

              //En on pressed agregamos una funcion que la creamos despues de las variables

              onPressed: tirarDados,

              //Agregamos estilo al boton

              style: TextButton.styleFrom(

                foregroundColor: Colors.white,

                textStyle: const TextStyle(

                  fontSize: 28,

                ),

                //padding: const EdgeInsets.only(top: 20),

              ),

              child: const Text('Tirar Dado'),

            ),

          ],

        ),

      ),

    );

  }

}

1. Introducción a los widgets con estado.

Ya tenemos nuestra aplicación casi terminada, falta la función que al presionar el botón “Tirar Dado”, me genere una imagen diferente a la que hemos programado hasta el momento y para ello, se necesita un widget con estado.

Un widget con estado en Flutter es un widget que puede cambiar su apariencia o comportamiento en respuesta a eventos o cambios en los datos. A diferencia de un widget sin estado, que se construye una vez y no cambia durante su vida útil, un widget con estado puede ser reconstruido varias veces durante su vida útil. Un widget con estado se crea extendiendo la clase StatefulWidget y proporcionando una implementación para el método createState, que devuelve una instancia de una subclase de State. La clase State contiene la lógica y el estado interno del widget y puede ser reconstruida cuando cambia el estado.

Ahora bien, nuestro código gradient\_container.dart es un widget sin estado, y necesitamos convertirlo a un widget con estado, por tal motivo debemos dividir el código en 2 partes, la parte que es estable que no cambia y la parte que si cambia; para ello, crearemos un archivo llamado “dice\_roller.dart”

Dentro de este archivo trasladaremos todo el código de gradient\_container.dart pero desde la parte de la columna.

El siguiente ya es el código de gradient\_container.dart:

import 'package:flutter/material.dart';

import 'package:flutter\_dados\_1/dice\_roller.dart';

//Creacion de variable para el inicio del gradiente

const startAlignment = Alignment.topLeft;

//Creacion de variable para el final del gradiente

const endAlignment = Alignment.bottomRight;

//Creación de un widget personalizado

class GradientContainer extends StatelessWidget {

  const GradientContainer(this.colorOne, this.colorTwo, {super.key});

  //Creamos las variables para los colores

  final Color colorOne;

  final Color colorTwo;

  @override

  Widget build(BuildContext context) {

    return Container(

      decoration: BoxDecoration(

        gradient: LinearGradient(

          //Agregamos las valiables colores

          colors: [

            colorOne,

            colorTwo,

          ],

          //agregar inicio y final del gradiente

          begin: startAlignment,

          end: endAlignment,

        ),

      ),

      child: const Center(

        child: DiceRoller(),

      ),

    );

  }

}

Y el siguiente es el código del archivo dice\_roller.dart:

import 'dart:math';

import 'package:flutter/material.dart';

final randomizer =

    Random(); //Creamos una variable para generar un numero aleatorio

class DiceRoller extends StatefulWidget {

  const DiceRoller({super.key});

  @override

  State<DiceRoller> createState() => \_DiceRollerState();

}

class \_DiceRollerState extends State<DiceRoller> {

  var currentDiceRoll = 2;

  //Creamos una funcion para generar un numero aleatorio y tirar los dados

  void tirarDados() {

    //Creamos una variable para generar un numero aleatorio

    setState(() {

      currentDiceRoll =

          randomizer.nextInt(6) + 1; //Genera un numero aleatorio entre 1 y 6

    });

    //Agregamos un setState para que se actualice la interfaz

  }

  @override

  Widget build(BuildContext context) {

    return Column(

      //Agregamos el main axis alignment para centrar el contenido

      mainAxisSize: MainAxisSize.min,

      //Como quieremos que la imagen y el texto esten centrados, los agrupamos en una lista dentro del widget Column

      children: [

        Image.asset(

          'assets/images/dice-$currentDiceRoll.png',

          width: 200,

        ),

        //Agregaremos una caja que ocupara un espacio dentro de la interfaz pero que no tendra contenido

        const SizedBox(

          height: 20,

        ),

        TextButton(

          //En on pressed agregamos una funcion que la creamos despues de las variables

          onPressed: tirarDados,

          //Agregamos estilo al boton

          style: TextButton.styleFrom(

            foregroundColor: Colors.white,

            textStyle: const TextStyle(

              fontSize: 28,

            ),

            //padding: const EdgeInsets.only(top: 20),

          ),

          child: const Text('Tirar Dado'),

        ),

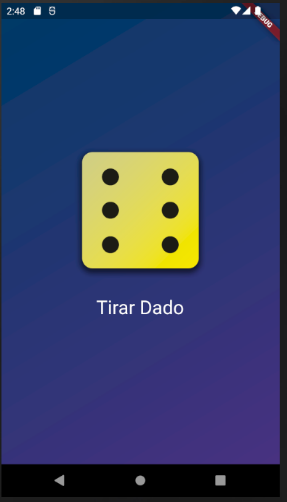
      ],

    );

  }

}

Con esto, ya nuestra aplicación ya esta funcionando, todo limpio y perfecto.

Lo que se ha logrado es una interfaz con la siguiente apariencia, en donde cada vez que presione el botón “Tirar Dato”, la imagen del dado cambie aleatoriamente…