

Creando nuestra primer App con Flutter

Diseñando y Desarrollando aplicaciones con Flutter



Índice de la guía

Ésta guía está basada en el tutorial “Write your First App “ que ofrece el sitio oficial de Flutter.dev.

Pasos:

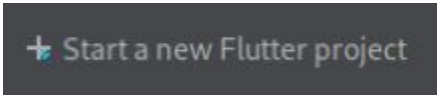
- Paso 1 - Crear el proyecto y una pantalla básica
- Paso 2 - Usar un paquete externo
- Paso 3 - Añadir un Stateful Widget
- Paso 4 - Crear una lista scrollable infinita



Paso 1 - Creando el proyecto

Para crear nuestro proyecto con Flutter, antes que nada deberemos tenerlo instalado, y si no fuera ese el caso, está la documentación en el repositorio para dejar todo listo para desarrollar.

Una vez que abrimos Android Studio, nos ofrecerá entre sus opciones:

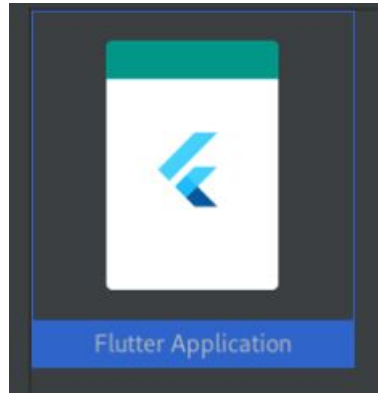


+ Start a new Flutter project

Haciendo click en este, comenzaremos la creación de nuestro nuevo proyecto.

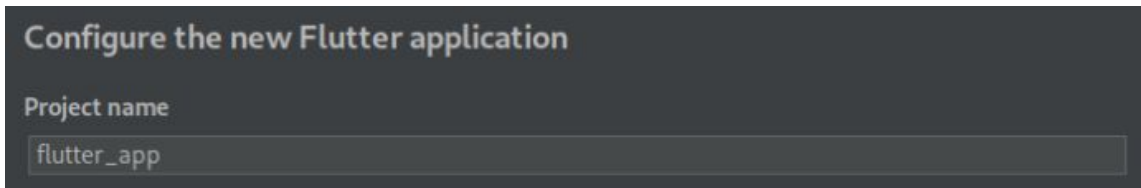
Paso 1 - Configurando el proyecto

Se nos abrirá una nueva pantalla en la cual deberemos seleccionar la siguiente opción.



Paso 1 - Configurando el proyecto

- Si apretamos siguiente deberíamos pasar a la página para configurar nombre y detalles del proyecto. En el campo del nombre de proyecto debe estar en **minúscula**. En ese campo colocamos el nombre que creamos conveniente

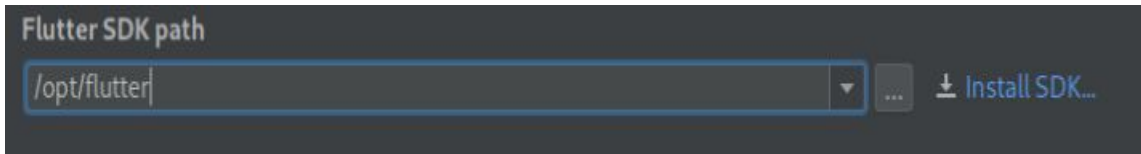


Configure the new Flutter application

Project name

flutter_app

- En el campo de Flutter SDK Path, si se encuentra vacío, deberemos seleccionar el directorio donde quedó instalado Flutter



Flutter SDK path

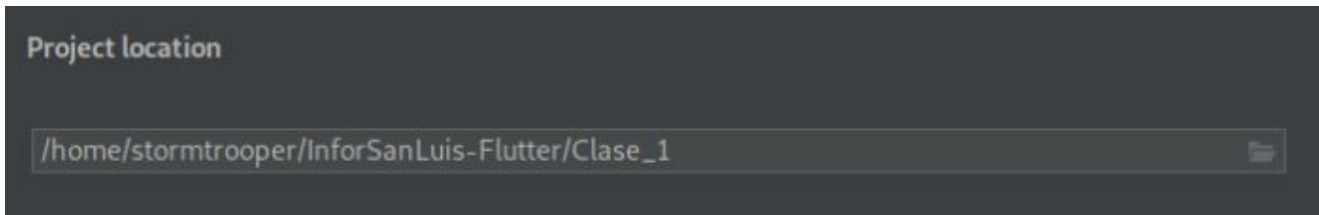
/opt/flutter

... [Install SDK...](#)



Paso 1 - Configurando el proyecto

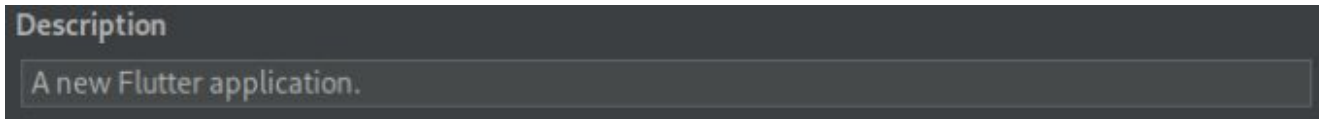
- También debemos indicar donde guardaremos nuestro proyecto.



Project location

/home/stormtrooper/InforSanLuis-Flutter/Clase_1

- Finalmente, podemos agregar una descripción donde agregue más información sobre nuestro proyecto.



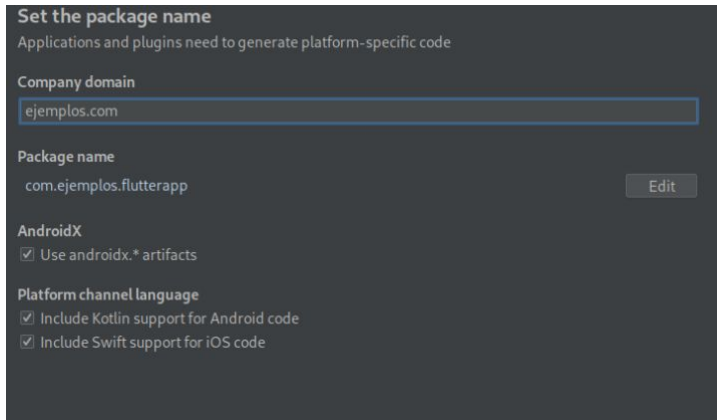
Description

A new Flutter application.



Paso 1 - Configurando el proyecto

Luego la pantalla final antes que podamos ver nuestro proyecto funcionando, podemos configurar el nombre del paquete, y si queremos incluir lenguajes específicos de las distintas plataformas. Dejando todo por defecto, es suficiente para los alcances del curso.



Set the package name
Applications and plugins need to generate platform-specific code

Company domain
ejemplos.com

Package name
com.ejemplos.flutterapp Edit

AndroidX
☒ Use androidx.* artifacts

Platform channel language
☒ Include Kotlin support for Android code
☒ Include Swift support for iOS code

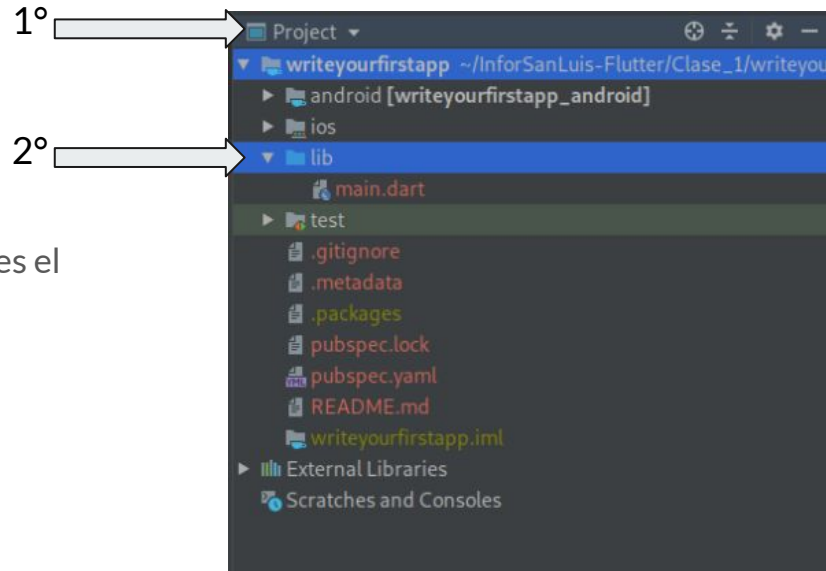
Paso 1 - Visualización del Proyecto

En la imagen que se ve, nos interesa

concentrarnos en tener configurado

en 1º, la vista como “Project”, y el directorio que nos interesa es el

“lib”, donde estará todo nuestro futuro código.





Paso 1 - main.dart

En este punto nuestro main.dart, debería tener algo como lo que sigue:

Éste código será reemplazado por uno más simple, para empezar

a darle forma a nuestra primer app.

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

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

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        // This is the theme of your application.
        // Try running your application with "flutter run". You'll see the
        // application has a blue toolbar. Then, without quitting the app, try
        // changing the primarySwatch below to Colors.green and then invoke
        // "hot reload" (press "r" in the console where you ran "flutter run",
        // or simply save your changes to "hot reload" in a Flutter IDE).
        // Notice that the counter didn't reset back to zero; the application
        // is not restarted.
        primarySwatch: Colors.blue,
      ), // ThemeData
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    ); // MaterialApp
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage(Key key, this.title) : super(key: key);

  // This widget is the home page of your application. It is stateful, meaning
  // that it has a State object (defined below) that contains fields that affect
  // how it looks.

  // This class is the configuration for the state. It holds the values (in this
  // case the title) provided by the parent (in this case the App widget) and
  // used by the build method of the State. Fields in a Widget subclass are
  // always marked "final".

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework that something has
      // changed in this State, which causes it to rerun the build method below
      // so that the display can reflect the updated values. If we changed
      // counter without calling setState(), then the build method would not be
      // called again, and so nothing would appear to happen.
      counter++;
    });
  }
}
```



Paso 1 - Pantalla básica

Lo primero que haremos será definir una pantalla básica, en el directorio del repositorio, en `/InforSanLuis-Flutter/Clases/Clase_1/Creando_Primer_App_Primer_Parte/Pasos_PrimerApp/Paso 1`, estará el código base, el cual utilizaremos para reemplazar el existente.



Paso 1 - Pantalla básica (Observaciones)

- En el ejemplo creamos una “**Material App**”. Material es un estilo de diseño de Google estándar de móvil y web.
- El método `main()` usa la **notación** (`=>`), ya que es una función de una sola línea y nos permite un código más limpio, que hacerlo de la manera tradicional.
- Ésta app extiende ***StatelessWidget*** que hace que la App sea un widget en sí. Acordarse de que en Flutter todo es un ***Widget***.
- El trabajo principal de un Widget es proveer el método ***build()*** que describe cómo mostrar el widget en función de los otros de menor nivel descriptos dentro de `build`.
- El cuerpo de este ejemplo consiste en un Widget ***Center***, que contiene un ***Text*** como hijo. Lo que hace el Center es alinear a su hijo, en este caso el Text, en el centro de la pantalla.

Paso 2 - Usar un paquete externo

En este paso se aprenderá a usar un paquete open-source que se llama *english_words*, donde este contiene miles de palabras en inglés, además de otras funciones.

- https://pub.dev/packages/english_words

El archivo de pubspec maneja todos los assets y dependencias para una aplicación de Flutter. En el *pubspec.yaml*, hay que añadir el paquete como dependencia.



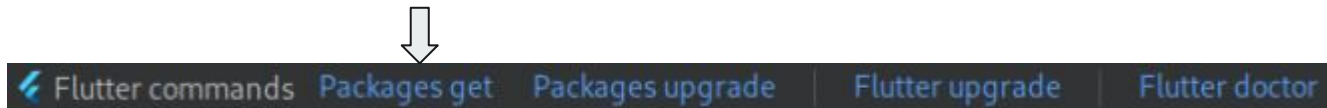
```
dependencies:
  flutter:
    sdk: flutter

# The following adds the Cupertino Icons font to your application.
# Use with the CupertinoIcons class for iOS style icons.
cupertino_icons: ^0.1.2
english_words: ^3.1.5
```



Paso 2 - Packages get

Apenas añadimos este paquete al *pubspec.yaml*, Android Studio nos va a ofrecer estos comandos, hacemos un **Packages get**, para que descargue la librería y podamos hacer uso de ella en el código.



Realizar esta acción genera un archivo llamado *pubspec.lock* que tiene una lista de todos los paquetes descargados al proyecto y sus versiones.



Paso 2 - Importando el nuevo paquete a la App

En *lib/main.dart*, importamos el paquete añadiendo la línea que está a continuación al principio del archivo.

```
import 'package:flutter/material.dart';  
import 'package:english_words/english_words.dart';
```

Con esto ya hemos logrado importar la librería que necesitábamos a nuestro código.



Paso 2 - Usando la nueva librería

Para lograr usarla, vamos a definir una variable constante en tiempo de ejecución (final) la cual le vamos a asignar una palabra random, haciendo uso de la clase ***WordPair.random()*** .

Luego en lugar de mostrar el ***Hello World*** que teníamos anteriormente, mostraremos esta palabra random.

Probarlo y cualquier duda, dejamos el código de ejemplo para validar el ejercicio.

/InforSanLuis-Flutter/Clases/Clase_1/Creando_Primer_App_Primer_Parte/Pasos_PrimerApp/Paso 2, estará el código base, con el cual validamos que estemos en la dirección correcta.



Paso 2 - Hot Reload

Si tenemos la aplicación corriendo ya sea en el celular físico o en el emulador, deberías estar visualizando que cada vez que lo aprietes, cambia la palabra mostrada.

Se debe a que cada vez que se redibuja la pantalla, una nueva palabra es generada, debido a que por detrás el método **build()** es llamado cada vez que una pantalla se redibuja.



Paso 3 - Añadiendo un Stateful Widget

Bueno hemos visto hasta ahora que los **Stateless Widget** son inmutables, lo que significa que sus propiedades no pueden cambiar o también es lo mismo que decir que sus valores son *final*.

Los **Stateful Widget** mantienen un estado que puede o no cambiar durante la vida del Widget. Para implementarlo en código requiere como básico dos clases:

- **StatefulWidget class** - donde esta crea una instancia de la **State**
- **State class**

******La clase **StatefulWidget** en sí es inmutable, pero la **State** persiste en la vida de la Stateful



Paso 3 - Creando el StatefulWidget

En este paso crearemos un StatefulWidget *RandomWords*, que crea su propia clase **State** - > **RandomWordState**. Usaremos entonces a *RandomWords* como hijo de la clase existente **MyApp** o como le hayamos llamado a nuestra clase principal (La que es llamada por **runApp()**.)



Paso 3 - Creando clase RandomWordsState

Para crear la RandomWordState, lo haremos de la siguiente manera:

```
class RandomWordsState extends State<RandomWords>{  
  //TODO: Add build() method  
}
```

La definimos, como vemos, extiende a **State**, donde lo que estamos diciendo es que usaremos la clase genérica State pero especializada para **RandomWords**. La mayoría de la lógica y estado va estar en ésta clase, ya que mantendrá el estado de **RandomWords**.

******Veremos que no da indicaciones en **rojo** ya que todavía no hemos definido la clase **RandomWords**.



Paso 3 - Añadiendo RandomWords a main.dart

Ahora para añadir *RandomWords* le indicamos que cuando cree el estado, sea del tipo que especificamos anteriormente *RandomWordsState*.

```
class RandomWords extends StatefulWidget{  
  @override  
  RandomWordsState createState() => RandomWordsState();  
}
```

Luego nos va a indicar error la clase *RandomWordsState* que no tenemos implementado el método *build()*.



Paso 3 - Implementando build()

En este paso añadiremos una implementación básica de `build()` donde generará el par de palabras *moviendo la generación de palabras* desde `MyApp` o el nombre de la clase principal donde se generaba la palabra a la clase `RandomWordsState`.

```
class RandomWordsState extends State<RandomWords>{  
  @override  
  Widget build(BuildContext context) {  
    final wordPair = WordPair.random();  
    return Text(wordPair.asPascalCase);  
  }  
}
```



Paso 3 - Utilizando RandomWords

Ahora en lugar de usar el `Text()` como hijo de `Center`, utilizaremos a `RandomWords`.

El paso simplemente consiste en cambiar el hijo de `Center` y cambiarlo por `RandomWords`.

```
body: Center(  
  child: RandomWords(),  
), // Center
```

Reiniciamos la App y deberíamos esperar el mismo comportamiento que antes, pero con el código actualizado hasta el momento.

En el directorio que está a continuación, tendrán un ejemplo del código para cumplir con ambos pasos 2 y 3 respectivamente.

/InforSanLuis-Flutter/Clases/Clase_1/Creando_Primer_App_Primer_Parte/Pasos_PrimerApp/Paso 2 y 3



Paso 4 - Creando una Lista Infinita

En este paso aprenderemos a generar y listar una lista de palabras que crece infinitamente.

A medida que el usuario "scrollee", la lista mostrada en un Widget llamado **ListView**, crecerá indefinidamente.

El constructor "**builder**" de la **Listview**, permite generar una lista de manera "perezosa", es decir, que la va creando a medida que lo va necesitando.



Paso 4 - Creando una Lista Infinita

- Para lograrlo, empezaremos a jugar con la clase *RandomWordsState*, deberemos agregar una variable que mantenga una Lista de *WordPair*, que es el tipo de objeto que devuelve *WordPair.random()*.
- Y además agregaremos un estilo de texto, para mostrar con una fuente de mayor tamaño las palabras en la lista.

```
final sugerencias = <WordPair>[];  
final fuenteLista = const TextStyle(fontSize: 18.0);
```




Paso 4 - Añadiendo constructor de Lista

En este paso añadiremos una función a ***RandomWordsState*** que retornará una lista a la que llamaremos ***_buildSugerencias()***. Este método construye la ***ListView*** que mostrará la lista con las palabras en ***_sugerencias***.

Como habíamos mencionado la ***ListView*** proporciona una propiedad de generador, ***itemBuilder***, que es un constructor tipo *factory* y una función callback especificada como una función anónima.

Se pasan dos parámetros a la función: ***BuildContext*** y el índice de iteración ***i***. El iterador comienza en 0 y se incrementa cada vez que se llama a la función.

Este modelo permite que la lista sugerida crezca infinitamente a medida que el usuario se desplaza.



Paso 4 - Añadiendo `_buildSugerencias()`

La función como vemos devuelve un Widget, donde este Widget será la ***ListView***. La función ***itemBuilder*** es llamada por cada palabra que haya en la lista ***_sugerencias*** que definimos anteriormente.

- `/* 1 */` Solo estamos añadiendo un padding entre la Lista y quien la contiene.
- `/* 2 */` En el caso que el índice llegue a la última palabra de ***_sugerencias*** generamos las nuevas 10 palabras a mostrar.
- `/* 3 */` Llamamos a una función que retorna un Widget que llamaremos al que construye cada ítem de la lista con cada palabra de la lista ***_sugerencias***.

** Las referencias están asociadas a las indicadas en el código de la filmina a continuación



Paso 4 - Añadiendo _buildSugerencias()

```
Widget _buildSugerencias() {  
  return ListView.builder(  
    padding: const EdgeInsets.all(16.0), /*1*/  
    itemBuilder: /*1*/ (context, i) {  
      if (i >= _sugerencias.length) {  
        _sugerencias.addAll(generateWordPairs().take(10)); /*2*/  
      }  
      return _buildPalabra(_sugerencias[i]); /*3*/  
    }); // ListView.builder  
}
```

Paso 4 - Añadiendo `_buildPalabra(WordPair palabra)`

La función como vemos devuelve un Widget, donde este Widget será el ítem con la palabra que le pasemos por parámetro, con un estilo que le daremos dentro de la misma función. La función `_buildPalabra()` es llamada por cada palabra que haya en la lista a medida que se va llamando *itemBuilder*.

El estilo que le daremos será armar un *ListTile*, con un *Text* como hijo el cual contendrá el texto pasado por parámetro, y le seguirá al *ListTile*, un *Divider*, para que se note la división entre los ítems.

- `/* 5 */` Definimos el Column, ya que tendremos primero el *ListTile*, y luego el *Divider*
- `/* 6 */` Utilizamos el parámetro de la función para mostrarla en el *Text*.
- `/* 7 */` Utilizamos el estilo de texto que definimos al principio, un *TextStyle* que está en la variable *fuentesLista*.
- `/* 8 */` Agregamos el *Divider* luego del *ListTile*.

** Las referencias están asociadas a las indicadas en el código de la filmina a continuación



Paso 4 - Añadiendo `_buildPalabra(WordPair palabra)`

```
Widget _buildPalabra(WordPair palabra) {  
  return Column(  
    children: <Widget>[  
      ListTile(  
        title: Text(  
          palabra.asPascalCase,  
          style: _fuenteLista,  
        ), // Text  
      ), // ListTile  
      Divider(),  
    ], // <Widget>[]  
  ); // Column  
}
```



Paso 4 - Primer App Funcionando !

En el directorio que está a continuación, tendrán un ejemplo del código para cumplir con todos los pasos para validar lo que hayan hecho.

/InforSanLuis-Flutter/Clases/Clase_1/Creando_Primer_App_Primer_Parte/Pasos_PrimerApp/Paso 4

- Link al Tutorial “Write your First App” : <https://flutter.dev/docs/get-started/codelab>

******Se va encontrar diferencias donde define el método `_buildSuggestions()`, debido a que ese código tiene en cuenta de no dibujar un Divider para el último Item, si no que solo dibujarlo entre ellos. Por una cuestión de tener una visión más clara del código, se armó esta versión de ese curso.