

# Primeros Pasos con Flutter

Por **Victor Manuel Merayo Álvarez** - 30 abril, 2019



## Sumario

- [1. Introducción](#)
- [2. Entorno](#)
- [3. ¿Qué es Flutter?](#)
- [4. Instalación](#)
- [5. Creación de un proyecto con Flutter](#)
- [6. Ejemplo práctico](#)
- [7. Conclusiones](#)

## 1. Introducción

He empezado a trastear con Flutter y me gustaría compartir con todos vosotros lo que he aprendido.

En este tutorial explicaremos brevemente qué es Flutter, veremos cómo se instala y realizaremos un ejemplo práctico de una simple aplicación desarrollada utilizando este Framework.

***Empezamos .....***

## 2. Entorno

Este tutorial está escrito utilizando el siguiente entorno:

- **Hardware:** Dell Inspiron 15' (2.50 GHz Intel Core i7, 16 GB DDR3 )
- **Sistema Operativo:** Ubuntu 18.04.2 LTS
- **Entorno de desarrollo:** Android Studio 3.3

### 3. ¿Qué es Flutter?

Flutter es un SDK de Google creado para desarrollar aplicaciones nativas multiplataformas, que permite a los desarrolladores crear aplicaciones iOS y Android utilizando exactamente el mismo código.

La principales características son:

- Permite crear aplicaciones nativas multiplataforma (como se indicó anteriormente).
- Framework Reactivo.
- El core está escrito en C++, por lo que es muy rápido.
- Utiliza como lenguaje de programación Dart. Dart es un lenguaje orientado a objetos con varias características útiles: mixins, generics, isolates y tipos opcionales estáticos.
- Motor propio de renderización basado en Skia (No utiliza el Web View de las aplicaciones híbridas ni los widgets que vienen en los dispositivos).
- Existen una gran cantidad de widgets propios listos para ser utilizados. Las aplicaciones en Flutter están compuestas por un árbol de widgets, que mantienen entre ellos una relación de padre-hijo.
- Tiene una funcionalidad llamada **Hot Reload** que permite realizar los cambios en caliente, sin necesidad de parar y arrancar la aplicación. Esto hace que la programación sea más productiva y con menos esperas.
- Más rápido y eficaz que React Native, Native Script, Ionic, ...

Todo en Flutter es un widget. Existen dos tipos de widget: **Stateful** (con estado) y **Stateless** (Sin estado). Los widgets **Stateful** permiten interacción por parte del usuario, pudiendo cambiar su estado y por lo tanto su apariencia en el UI.

Cuando hablamos del estado de un widget (**State**), nos estamos refiriendo a valores que pueden cambiar, como por ejemplo un slider, un checkbox, ....

Sin más preámbulos pasemos a la faena, para ver realmente cómo podemos hacer aplicaciones con Flutter.

### 4. Instalación

Los pasos para la instalación de Flutter, en función del sistema operativo que queramos utilizar, los podemos encontrar en la siguiente página <https://flutter.dev/docs/get-started/install>.

En nuestro caso vamos a utilizar Linux.

Instalación del SDK de Flutter

1. Descargamos la última versión estable de Flutter

```
flutter_linux_v1.2.1-stable.tar.xz
```

2. Descomprimos el fichero en la ubicación donde lo queremos instalar

```
$ tar xf flutter_linux_v1.2.1-stable.tar.xz
```

3. Añadimos Flutter a nuestro PATH

```
$ export PATH="$PATH:`pwd`/flutter/bin"
```

4. Ejecutamos Flutter doctor para verificar si existe alguna dependencia necesaria que aún no tenemos instalada.

```
$ flutter doctor
```

El resultado de esta ejecución mostrará la siguiente información:

```
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel stable, v1.2.1, on Linux, locale es_ES.UTF-8)
    ✗ Downloaded executables cannot execute on host.
      See https://github.com/flutter/flutter/issues/6207 for more information
      On Debian/Ubuntu/Mint: sudo apt-get install lib32stdc++6
      On Fedora: dnf install libstdc++.i686
      On Arch: pacman -S lib32-libstdc++5 (you need to enable multilib:
        https://wiki.archlinux.org/index.php/Official_repositories#multilib)
[!] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
    ✗ Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses
[!] Android Studio (version 3.3)
    ✗ Flutter plugin not installed; this adds Flutter specific functionality.
    ✗ Dart plugin not installed; this adds Dart specific functionality.
[!] Connected device
! Doctor found issues in 4 categories.
```

Como podemos observar, flutter doctor nos informa sobre las dependencias pendientes de configurar.

Instalamos la librería **lib32stdc++6** de 32 bits que nos solicita, ya que estamos en un entorno de 64 bits.

```
$ sudo apt-get install lib32stdc++6
```

Ejecutamos el siguiente comando para aceptar todas las licencias solicitadas por el SDK

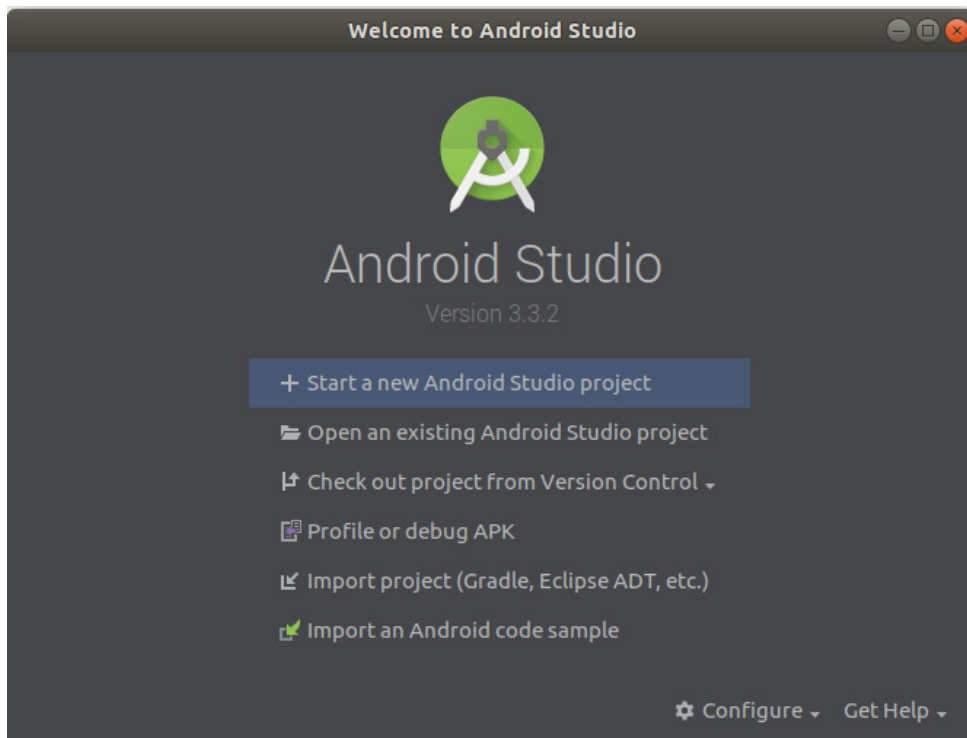
```
$ flutter doctor --android-licenses
```

Una vez aceptadas todas las licencias se mostrará el siguiente mensaje en pantalla

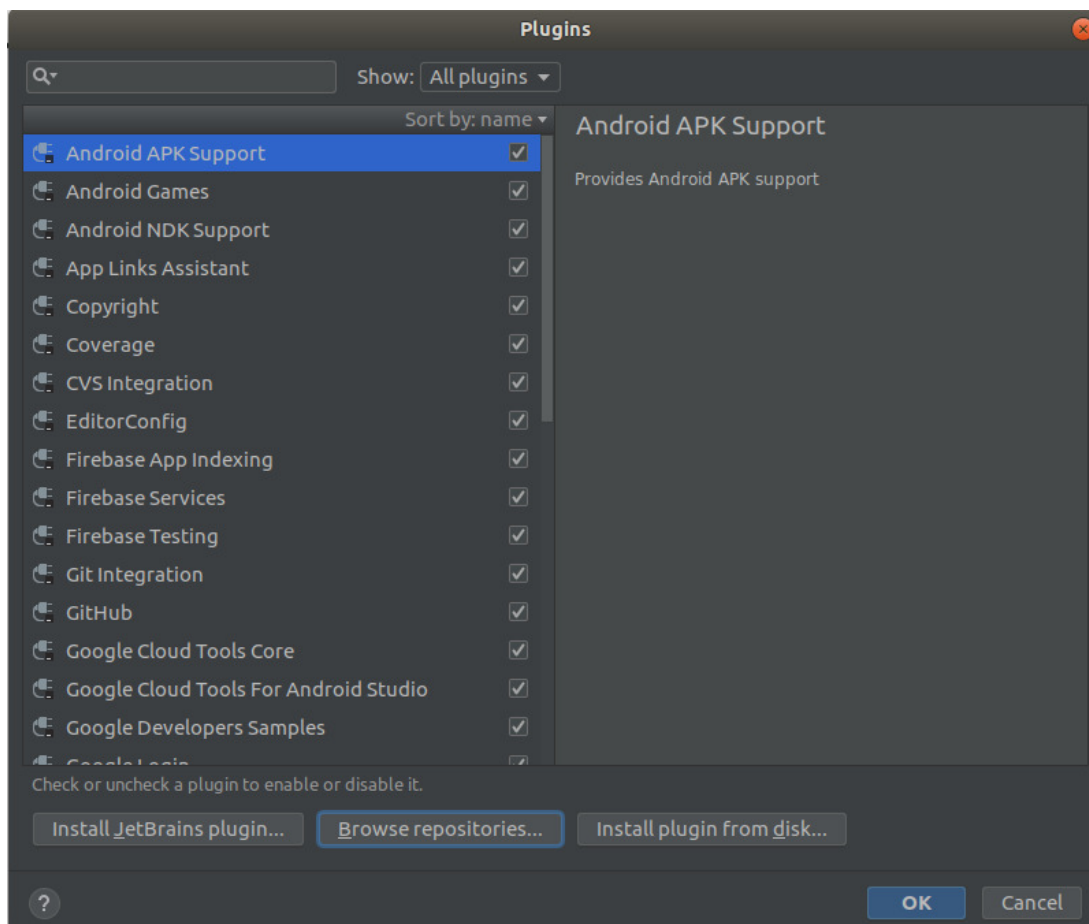
```
All SDK package licenses accepted
```

Instalamos en Android Studio los Plugins indicados (Flutter y Dart).

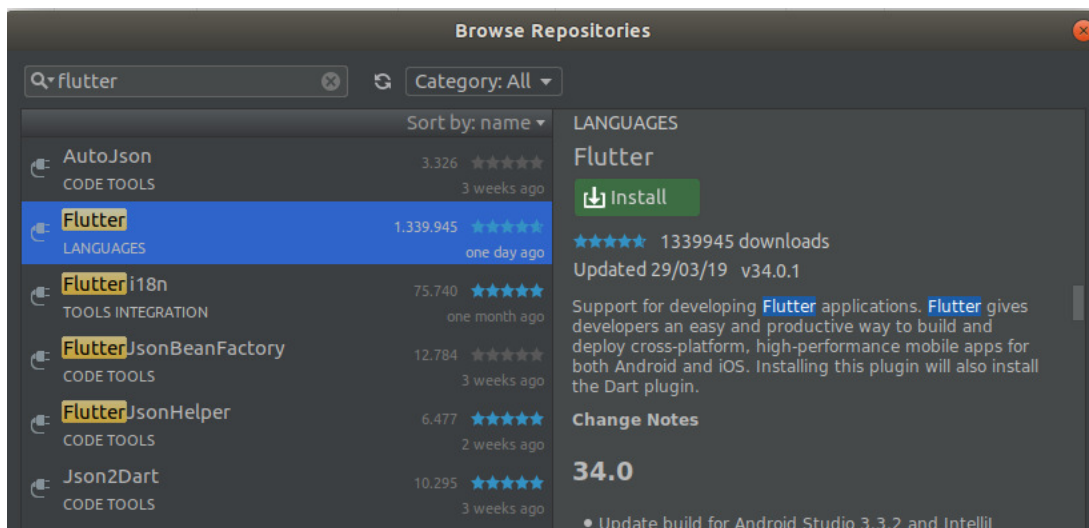
Para ello abrimos el Android Studio.



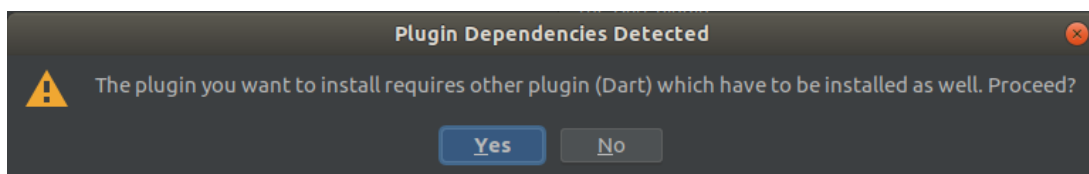
Hacemos click en **"Configure"** → **"Plugins"**



Haciendo click en **"Browse repositories"**, buscamos el Plugin para Flutter, lo seleccionamos y lo instalamos.

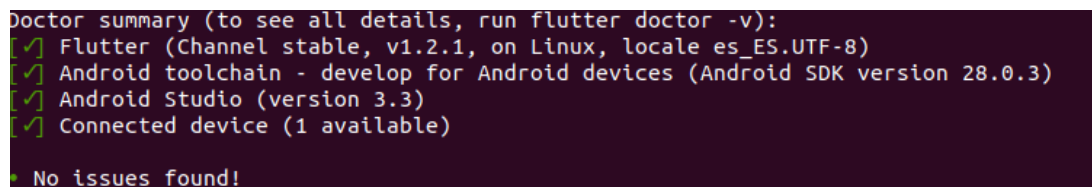


Cuando instalamos el Plugin de Flutter, nos indicará que necesitamos una dependencia con otro Plugin, en este caso será el de **Dart**, que tendremos que instalar también.



Si aceptamos, tendremos los dos Plugins instalados.

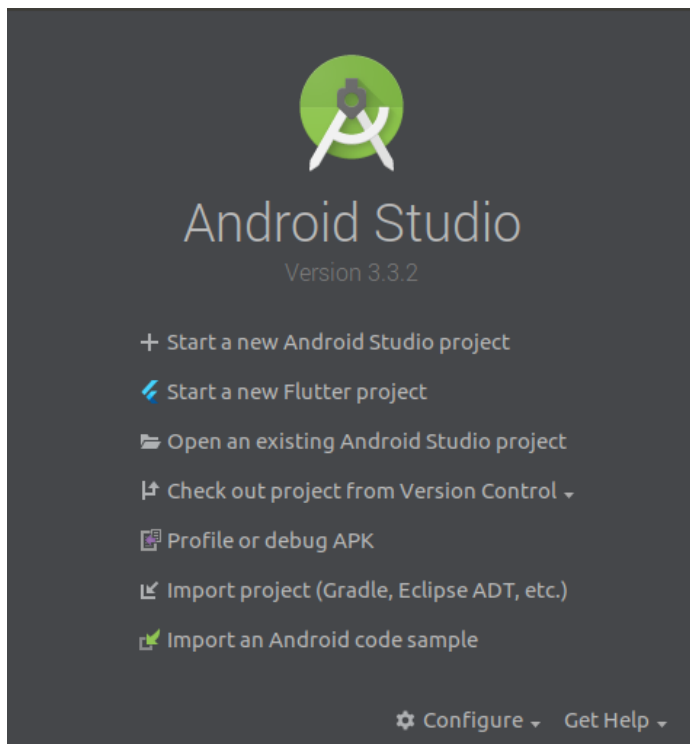
Volveremos a ejecutar el flutter doctor y deberíamos tener todo correcto



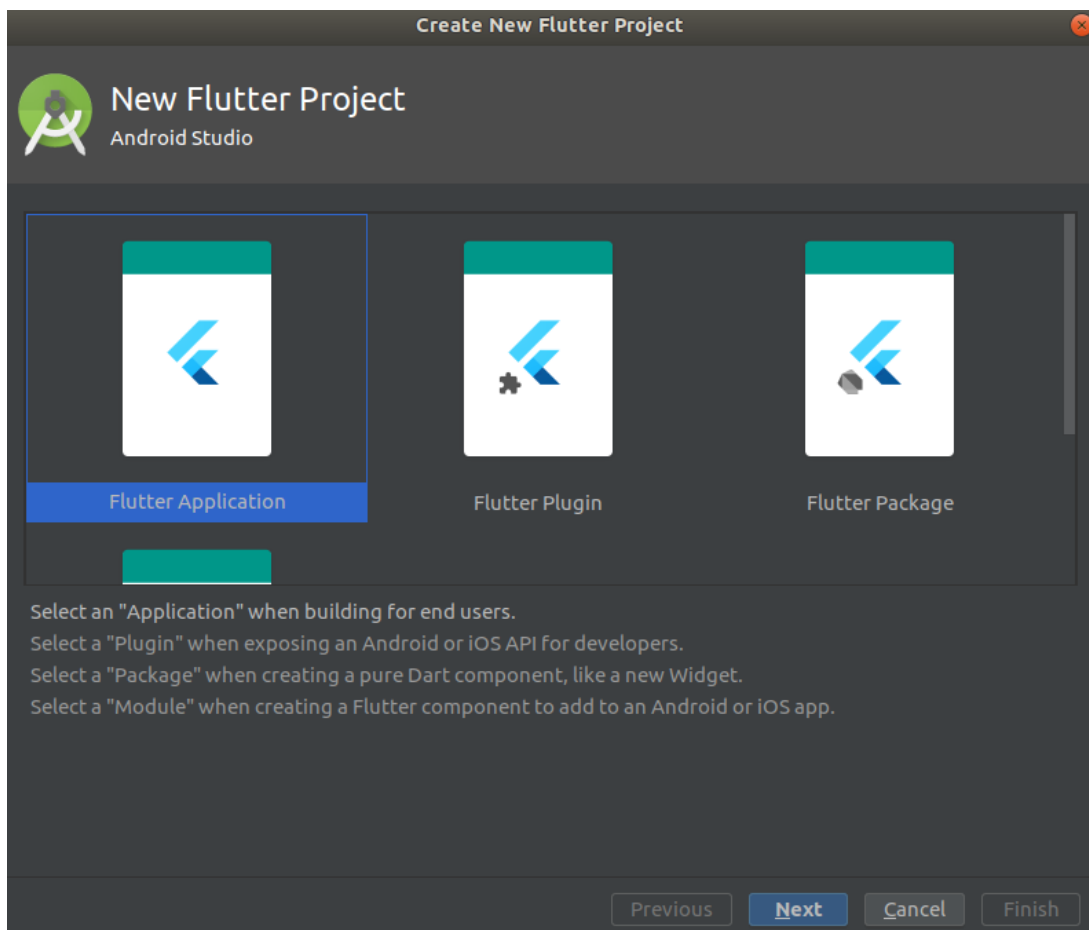
## 5. Creación de un Proyecto con Flutter

Vamos a mostrar cómo crear una primera aplicación en Flutter desde Android Studio. El proceso es muy sencillo y solo tenemos que seguir los siguientes pasos:

Abrimos Android Studio y en la pantalla principal seleccionamos **“Start a new Flutter project”**




En la siguiente pantalla seleccionamos que queremos crear una Aplicación Flutter.





Le indicamos el nombre del proyecto. En este caso dejamos el nombre por defecto que nos da Android Studio **"flutter\_app"**.


Create New Flutter Project

 **New Flutter Application**  
Android Studio

**Configure the new Flutter application**

**Project name**

**Flutter SDK path**  
   [↓ Install SDK...](#)

**Project location**  
 


**Description**

☐ Create project offline

[Previous](#) [Next](#) [Cancel](#) [Finish](#)

Seleccionamos el paquete principal de nuestro proyecto. Del mismo modo que en el caso anterior, dejaremos el nombre por defecto que nos da Android Studio.

Create New Flutter Project

 **New Flutter Application**  
Android Studio

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

**Company domain**

**Package name**  
 [Edit](#)

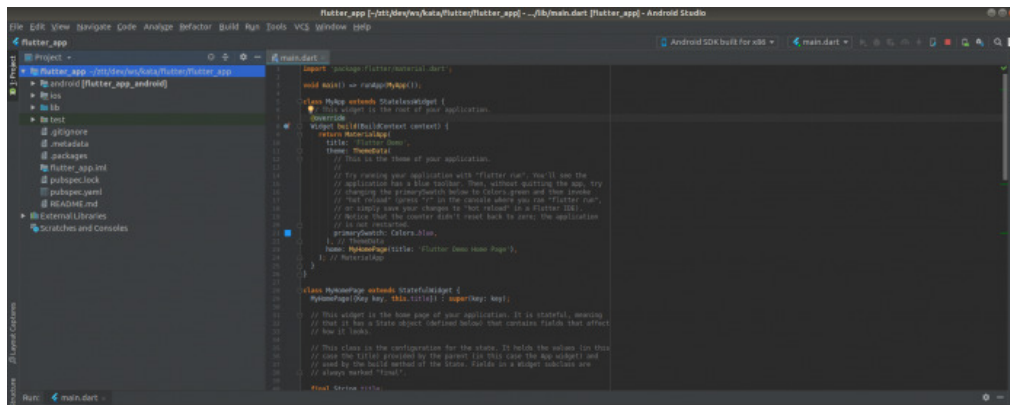
**Sample Application**  
☐ generate sample content:

**Platform channel language**  
☐ Include Kotlin support for Android code  
☐ Include Swift support for iOS code

[Previous](#) [Next](#) [Cancel](#) [Finish](#)

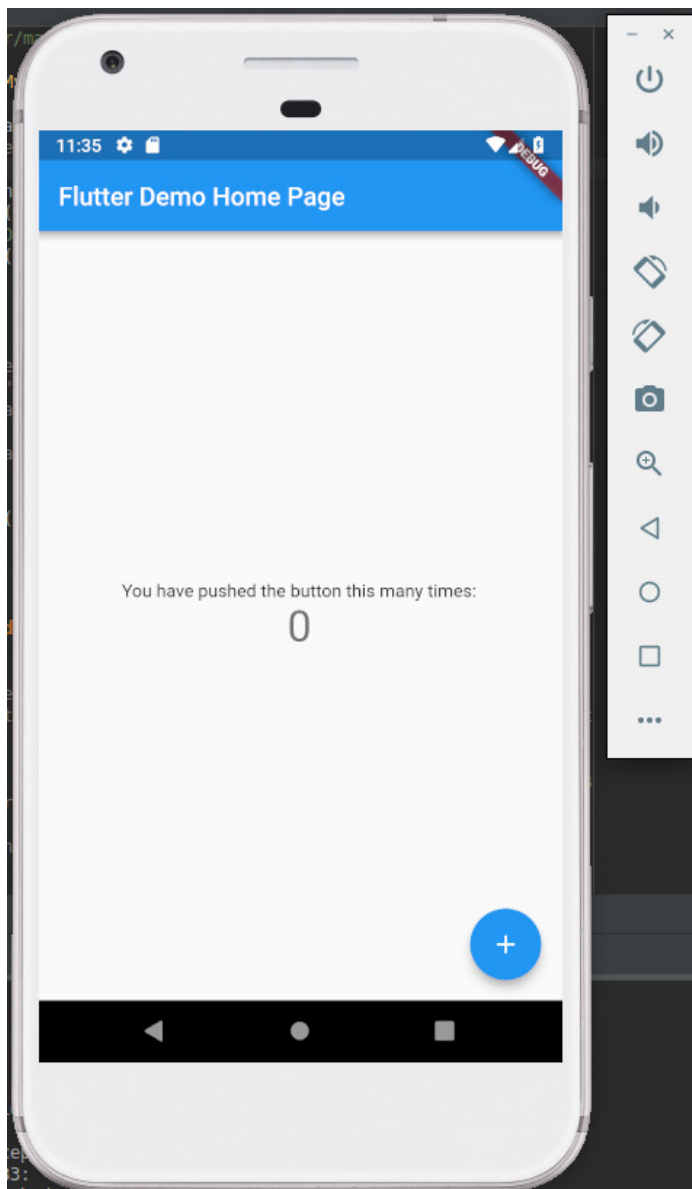


Pulsamos **"Finish"** y ya habremos creado nuestra primera aplicación en Flutter.



Si ahora lanzamos la aplicación en nuestro emulador, tendremos ejecutando nuestra primera aplicación Flutter.

## 6. Ejemplo Práctico

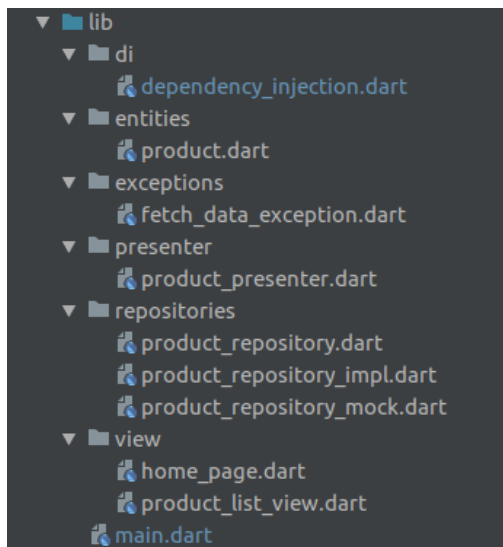


En este apartado partiremos de una aplicación Flutter que he desarrollado como ejemplo y cuyo código lo podéis descargar de mi github <https://github.com/vtcmer/flutter-product.git>.



La aplicación es muy sencilla, mostrará un listado de productos. Podremos configurar la aplicación para que arranque simulando un entorno productivo o un entorno de desarrollo con un mock.

Estructura del proyecto:



En la carpeta **lib** del proyecto encontraremos todo el código implementado.

Ahora pasaré a explicar cada una de las parte de este código.

1. **main.dart**: Clase principal que se encarga de lanzar la aplicación.

Como comentamos anteriormente, para crear aplicaciones en Flutter tenemos que utilizar Dart. Todos los proyectos Dart comienza con una función main que se encarga de iniciar la aplicación.

Creamos nuestro propio widget de tipo **StatelessWidget** (**MyApp**, ocupará el 100% de la pantalla), que será el que pasamos por parámetro al método **runApp**. En la propiedad **home** le indicaremos cual será el widget que se corresponderá con el UI principal de nuestra aplicación.

```
void main() async{
  Injector.configure(Environment.MOCK);
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      theme: new ThemeData(
        primarySwatch: Colors.blue,
        primaryColor: defaultTargetPlatform == TargetPlatform.iOS
          ? Colors.grey[100]
          : null
      ), // ThemeData
      home: new HomePage(),
    ); // MaterialApp
  }
}
```

El estado de un widget se guarda en un objeto de tipo **State** (será nuestra clase **HomePageState** que veremos más adelante); de esta manera se mantiene separado el

estado del widget de su apariencia. Cuando un objeto cambia de estado, el objeto que extiende de la clase **State** llama a **setState()** y es entonces cuando el widget se actualiza.

## 2. **View**: Vista de la aplicación.

En este caso tendremos:

- **product\_list\_view.dart**: Se trata de una interfaz a modo de Callback. Su objetivo es proporcionar la firma de los métodos que se encargarán de renderizar los datos en el UI y realizar las operaciones correspondientes en caso de un error (por ejemplo mostrar un mensaje identificando dicho error).

```
abstract class ProductListView{  
  
    void onLoadProductComplete(List<Product> productList);  
    void onLoadProductError();  
  
}
```

- **home\_page.dart**: UI de la aplicación que renderizará el listado de productos. Será un widget de tipo **StatefulWidget**.

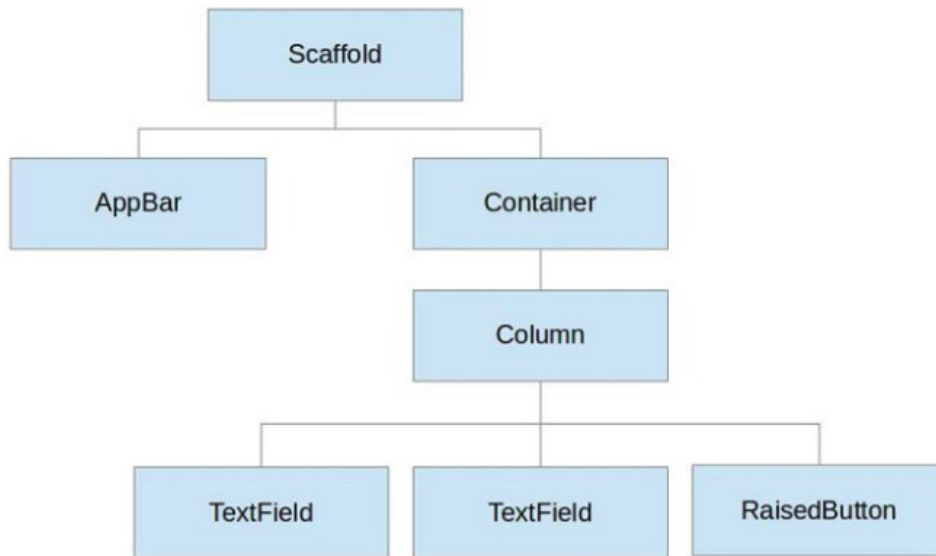
```
class HomePage extends StatefulWidget {  
  @override  
  _HomePageState createState() => new _HomePageState();  
}  
  
class _HomePageState extends State<HomePage> implements ProductListView{  
  
  ProductPresenter _productPresenter;  
  List<Product> _productList = [];  
  bool _isLoading = false;  
  
  final List<MaterialColor> _colors = [Colors.blue, Colors.indigo, Colors.red];  
  
  _HomePageState() {  
    this._productPresenter = new ProductPresenter(this);  
  }  
  
  @override  
  void initState() {...}  
  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(...); // Scaffold  
  }  
  
  Widget _productsWidget(){  
    return new Container(...); // Container  
  }  
  ListTile _getItem(Product item, MaterialColor color){...}  
  
  @override  
  void onLoadProductComplete(List<Product> items) {...}  
  
  @override  
  void onLoadProductError() {...}  
}
```

El fichero **home\_page.dart**, internamente contendrá:

- Una clase **HomePage** que es un widget de tipo **StatefulWidget** (con estado), que se actualizará con los productos que recuperemos de nuestro repositorio.
- Un objeto de tipo **State** (**\_HomePageState**). Ya que el widget principal es un widget con estado, tenemos que implementar esta clase.

En la clase **\_HomePageState** vamos a implementar los siguiente métodos:

- **initState:** Este método se invoca cuando es creado el estado e insertado el objeto en el árbol de widgets. Aprovechamos este punto para realizar una petición al presentador y solicitar el listado de productos.
- **build:** Devuelve el widget a renderizar. Este widget es de tipo Scaffold, que tiene una estructura predefinida



- **\_productWidget:** Devuelve un widget de tipo Contenedor con el listado de productos a mostrar.
  - **\_getItem:** Devuelve un widget especial de tipo **ListTile** para renderizar cada uno de nuestros productos.
3. **entities:** Este paquete contendrá una única entidad (**dart**) que será la representación de nuestro modelo de datos. En ese caso será un producto, que contendrá un identificador interno y un nombre.

```

class Product {
  int id;
  String name;
  Product(this.id, this.name);
}
  
```

4. **exceptions:** Paquete que contendrá las excepciones de la aplicación. En este caso sólo tendremos una excepción para capturar un error en la operación de búsqueda de productos.

```

class FetchDateException implements Exception{
  final String _message;
  FetchDateException([this._message]);

  String toString(){
    if (_message == null){
      return "Exception";
    } else {
      return "Exception: ${_message}";
    }
  }
}
  
```

5. **presenter**: Vamos a implementar un patrón de arquitectura **MVP** (modelo-vista-presentador) y en este paquete estará la clase que se encarga de solicitar los datos al repositorio y enviarlos a la vista.

```
class ProductPresenter{
  ProductListView _view;
  ProductRepository _productRepository;

  ProductPresenter(this._view){
    this._productRepository = new Injector().productRepository;
  }

  void loadProducts(){
    this._productRepository
      .fetchProduct()
      .then((result)=> this._view.onLoadProductComplete(result))
      .catchError((onError) => this._view.onLoadProductError());
  }
}
```

6. **repositories**: Este paquete contiene las clases encargadas de realizar el acceso a los datos. Ya que se trata de un pequeño proyecto de ejemplo, accederemos directamente desde el presentador al repositorio y nos saltaremos la capa de negocio.

Tendremos tres clases:

- **product\_repository.dart (ProductRepository)**: Interfaz con las firmas de los métodos que tendrán que implementar nuestras concreciones.

```
abstract class ProductRepository{
  Future<List<Product>> fetchProduct();
}
```

- **product\_repository\_mock.dart (ProductRepositoryMock)**: Implementación de nuestra interfaz que se utilizará como mock.

```
class ProductRepositoryMock implements ProductRepository{
  @override
  Future<List<Product>> fetchProduct() {...}
}
```

- **product\_repository\_impl.dart (ProductRepositoryImpl)**: Implementación de nuestra interfaz que se utilizará en el entorno productivo. Debería implementarse un acceso a base de datos o servicio Rest, pero para simplificar el ejemplo, se devolverán los datos desde una lista (*Lo podremos mejorar en futuras entradas* :)).

```
class ProductRepositoryImpl implements ProductRepository{
  @override
  Future<List<Product>> fetchProduct() async {...}
}
```

7. **di**: Este paquete contiene la clase **dart**, que como su propio nombre indica, se encarga de realizar la inyección de dependencias. En función del entorno establecido cuando se arranca la aplicación (**main.dart**), esta clase deberá identificar qué instancia de nuestro repositorio utilizar.

```
enum Environment{MOCK, PROD}

//DI
class Injector {

    static final Injector _singleton = new Injector._internal();

    static Environment _env;

    static void configure(Environment flavor){
        _env = flavor;
    }

    factory Injector(){
        return _singleton;
    }

    Injector._internal();

    /**
     * Recuperación del repositorio en función del entorno
     */
    ProductRepository get productRepository{
        switch(_env){
            case Environment.MOCK : return new ProductRepositoryMock();
            default: return new ProductRepositoryImpl();
        }
    }
}

}
```

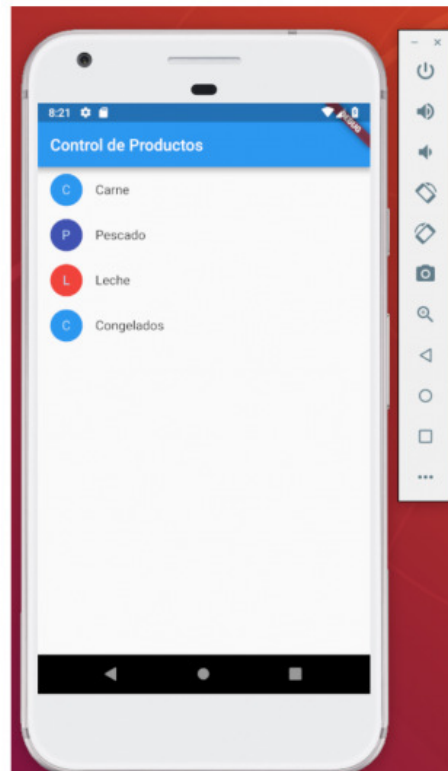
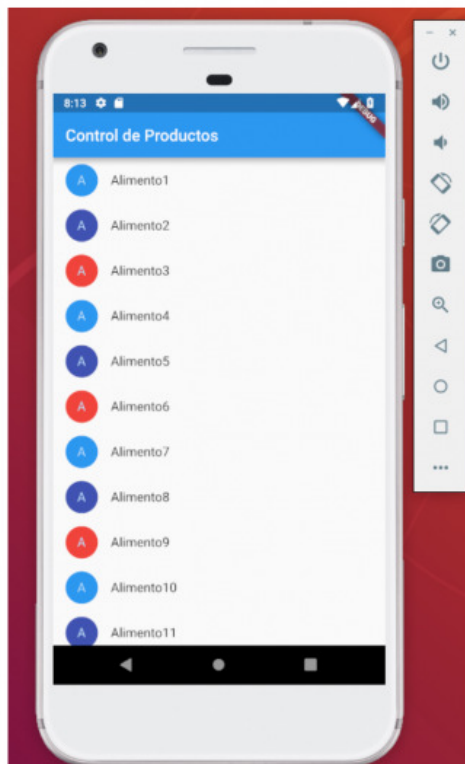
Arrancamos la aplicación para ver el resultado con cada uno de los entorno que tenemos configurados:

```
void main() async{
    Injector.configure(Environment.MOCK);
    runApp(new MyApp());
}

class MyApp extends StatelessWidget {...}
```

```
void main() async{
    Injector.configure(Environment.PROD);
    runApp(new MyApp());
}

class MyApp extends StatelessWidget {...}
```



## 7. Conclusiones

Hemos visto los primeros pasos con Flutter, cómo instalarlo y hacer una aplicación básica para ir tomando contacto con este nuevo Framework de desarrollo. Cada vez van surgiendo más y más widget que facilitan los desarrollos de nuevas funcionalidades.

Flutter (Flutter + Dart) será el Framework oficial de desarrollo para el nuevo sistema operativo que está desarrollando Google (**Fuchsia**), con lo que parece que tiene un futuro prometedor.

En próximas entradas trataré de mostrar nuevas funcionalidades, para ver las muchas posibilidades que nos puede ofrecer este Framework de desarrollo.

A continuación os dejo algunos enlaces de interés relacionados:

<https://flutter-es.io/docs>

<https://www.dartlang.org/guides/language/language-tour>

<https://material.io/develop/flutter/>

[https://en.wikipedia.org/wiki/Google\\_Fuchsia](https://en.wikipedia.org/wiki/Google_Fuchsia)



Esta obra está licenciada bajo [licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

Victor Manuel Merayo Álvarez

Responsable de Desarrollo en una empresa del sector de la Construcción

Puedes contactar conmigo:

Correo: [vtcmer@gmail.com](mailto:vtcmer@gmail.com)

LinkedIn: <https://www.linkedin.com/in/victor-manuel-merayo-álvarez-444b6a98>

Twitter: [@vmeralv](https://twitter.com/vmeralv)