

The background is a red Flutter logo. Overlaid on the logo are several lines of code in a monospaced font, with some characters in yellow and pink. The code includes file paths, file operations, and fingerprinting logic.

FLUTTER

www.postparaprogramadores.com

Contenido

Flutter - Inicio

Flutter - Introducción

Flutter - Instalación

Crear una aplicación simple en Android Studio

Flutter - Aplicación de arquitectura

Introducción a la programación de dardos

Flutter - Introducción a los widgets

Flutter - Introducción a los diseños

Flutter - Introducción a los gestos

Flutter - Gestión del Estado

Flutter - Animación

Flutter - Escribir código específico de Android

Flutter - Escribir código específico de IOS

Flutter - Introducción al paquete

Flutter - Accediendo a la API REST

Flutter - Conceptos de base de datos

Flutter - Internacionalización

Flutter - Pruebas

Flutter - Despliegue

Flutter - Herramientas de desarrollo

Flutter: escritura de aplicaciones avanzadas

Flutter - Conclusión

Descarga más libros de programación GRATIS [click aquí](#)



**Síguenos en Instagram para que estés al tanto de los
nuevos libros de programación. [Click aqui](#)**

Flutter - Introducción

En general, desarrollar una aplicación móvil es una tarea compleja y desafiante. Hay muchos marcos disponibles para desarrollar una aplicación móvil. Android proporciona un marco nativo basado en el lenguaje Java e iOS proporciona un marco nativo basado en el lenguaje Objective-C / Swift.

Sin embargo, para desarrollar una aplicación que soporte ambos sistemas operativos, necesitamos codificar en dos idiomas diferentes usando dos marcos diferentes. Para ayudar a superar esta complejidad, existen marcos móviles que admiten ambos sistemas operativos. Estos marcos van desde un simple marco de aplicación móvil híbrido basado en HTML (que usa HTML para la interfaz de usuario y JavaScript para la lógica de la aplicación) hasta un marco complejo de lenguaje específico (que hace el trabajo pesado de convertir código a código nativo). Independientemente de su simplicidad o complejidad, estos marcos siempre tienen muchas desventajas, uno de los principales inconvenientes es su bajo rendimiento.

En este escenario, Flutter, un marco simple y de alto rendimiento basado en el lenguaje Dart, proporciona un alto rendimiento al representar la IU directamente en el lienzo del sistema operativo en lugar de hacerlo a través del marco nativo.

Flutter también ofrece muchos widgets (UI) listos para usar para crear una aplicación moderna. Estos widgets están optimizados para entornos móviles y diseñar la aplicación usando widgets es tan simple como diseñar HTML.

Para ser específicos, la aplicación Flutter es en sí misma un widget. Los widgets Flutter también admiten animaciones y gestos. La lógica de la aplicación se basa en la programación reactiva. El widget puede tener opcionalmente un estado. Al cambiar el estado del widget, Flutter comparará automáticamente (programación reactiva) el estado del widget (antiguo y nuevo) y renderizará el widget solo con los cambios necesarios en lugar de volver a representar todo el widget.

Discutiremos la arquitectura completa en los próximos capítulos.

Características del Flutter

Flutter Framework ofrece las siguientes características a los desarrolladores:

- Marco moderno y reactivo.
- Utiliza el lenguaje de programación Dart y es muy fácil de aprender.
- Desarrollo rápido.
- Interfaces de usuario hermosas y fluidas.
- Enorme catálogo de widgets.
- Ejecuta la misma interfaz de usuario para múltiples plataformas.
- Aplicación de alto rendimiento.

Ventajas de Flutter

Flutter viene con widgets hermosos y personalizables para un alto rendimiento y una excelente aplicación móvil. Cumple con todas las necesidades y requisitos personalizados. Además de estos, Flutter ofrece muchas más ventajas como se menciona a continuación:

- Dart tiene un gran repositorio de paquetes de software que le permite ampliar las capacidades de su aplicación.
- Los desarrolladores necesitan escribir solo una base de código único para ambas aplicaciones (plataformas Android e iOS). *Flutter* puede extenderse a otra plataforma también en el futuro.
- Flutter necesita menos pruebas. Debido a su base de código único, es suficiente si escribimos pruebas automatizadas una vez para ambas plataformas.
- La simplicidad de Flutter lo convierte en un buen candidato para un desarrollo rápido. Su capacidad de personalización y extensibilidad lo hacen aún más poderoso.
- Con Flutter, los desarrolladores tienen control total sobre los widgets y su diseño.
- Flutter ofrece excelentes herramientas para desarrolladores, con una increíble recarga en caliente.

Desventajas de Flutter

A pesar de sus muchas ventajas, el Flutter tiene los siguientes inconvenientes:

- Dado que está codificado en lenguaje Dart, un desarrollador necesita aprender un nuevo idioma (aunque es fácil de aprender).
- El marco moderno intenta separar la lógica y la interfaz de usuario tanto como sea posible, pero, en Flutter, la interfaz de usuario y la lógica se entremezclan. Podemos superar esto usando una codificación inteligente y un módulo de alto nivel para separar la interfaz de usuario y la lógica.
- Flutter es otro marco para crear aplicaciones móviles. Los desarrolladores están teniendo dificultades para elegir las herramientas de desarrollo adecuadas en un segmento muy poblado.

Flutter - Instalación

Este capítulo lo guiará a través de la instalación de Flutter en su computadora local en detalle.

Instalación en Windows

En esta sección, veamos cómo instalar *Flutter SDK* y sus requisitos en un sistema Windows.

Paso 1 : vaya a URL, <https://flutter.dev/docs/get-started/install/windows> y descargue el último SDK de Flutter. A partir de abril de 2019, la versión es 1.2.1 y el archivo es flutter_windows_v1.2.1-stable.zip.

Paso 2 : descomprima el archivo zip en una carpeta, diga C: \ flutter \

Paso 3 : actualice la ruta del sistema para incluir el directorio de flutter bin.

Paso 4 : Flutter proporciona una herramienta, doctor de flutter para verificar que se cumplan todos los requisitos del desarrollo de flutter.

```
flutter doctor
```

Paso 5 : ejecutar el comando anterior analizará el sistema y mostrará su informe como se muestra a continuación:

```
Doctor summary (to see all details, run flutter doctor -v):  
[✓] Flutter (Channel stable, v1.2.1, on Microsoft Windows  
[Version  
10.0.17134.706], locale en-US)  
[✓] Android toolchain - develop for Android devices (Android  
SDK version  
28.0.3)  
[✓] Android Studio (version 3.2)  
[✓] VS Code, 64-bit edition (version 1.29.1)  
[!] Connected device  
! No devices available  
! Doctor found issues in 1 category.
```

El informe dice que todas las herramientas de desarrollo están disponibles pero el dispositivo no está conectado. Podemos solucionar esto conectando un dispositivo Android a través de USB o iniciando un emulador de Android.

Paso 6 : instale el SDK de Android más reciente, si lo informa flutter doctor

Paso 7 : instale el último Android Studio, si lo informa flutter doctor

Paso 8 : inicie un emulador de Android o conecte un dispositivo Android real al sistema.

Paso 9 : instala el complemento Flutter and Dart para Android Studio. Proporciona una plantilla de inicio para crear una nueva aplicación Flutter, una opción para ejecutar y depurar la aplicación Flutter en el propio estudio de Android, etc.

- Abra Android Studio.
- Haga clic en Archivo → Configuración → Complementos.
- Seleccione el complemento Flutter y haga clic en Instalar.
- Haga clic en Sí cuando se le solicite que instale el complemento Dart.
- Reinicia el estudio de Android.

Instalación en MacOS

Para instalar Flutter en MacOS, deberá seguir los siguientes pasos:

Paso 1 : vaya a URL, <https://flutter.dev/docs/get-started/install/macos> y descargue el último SDK de Flutter. A partir de abril de 2019, la versión es 1.2.1 y el archivo es flutter_macos_v1.2.1- stable.zip.

Paso 2 : descomprima el archivo zip en una carpeta, por ejemplo, / ruta / a / flutter

Paso 3 : actualice la ruta del sistema para incluir el directorio bin del Flutter (en el archivo `~ / .bashrc`).

```
> export PATH = "$PATH:/path/to/flutter/bin"
```

Paso 4 : habilite la ruta actualizada en la sesión actual utilizando el siguiente comando y luego verifíquela también.

```
source ~/.bashrc
source $HOME/.bash_profile
echo $PATH
```

Flutter proporciona una herramienta, doctor de flutter para verificar que se cumplan todos los requisitos del desarrollo de flutter. Es similar a la contraparte de Windows.

Paso 5 : instale el último XCode, si lo informa flutter doctor

Paso 6 : instale el último SDK de Android, si lo informa flutter doctor

Paso 7 : instale el último Android Studio, si lo informa flutter doctor

Paso 8 : inicie un emulador de Android o conecte un dispositivo Android real al sistema para desarrollar una aplicación de Android.

Paso 9 : abra el simulador de iOS o conecte un dispositivo de iPhone real al sistema para desarrollar una aplicación de iOS.

Paso 10 : instala el complemento Flutter and Dart para Android Studio. Proporciona la plantilla de inicio para crear una nueva aplicación de Flutter, opción para ejecutar y depurar la aplicación de Flutter en el propio estudio de Android, etc.

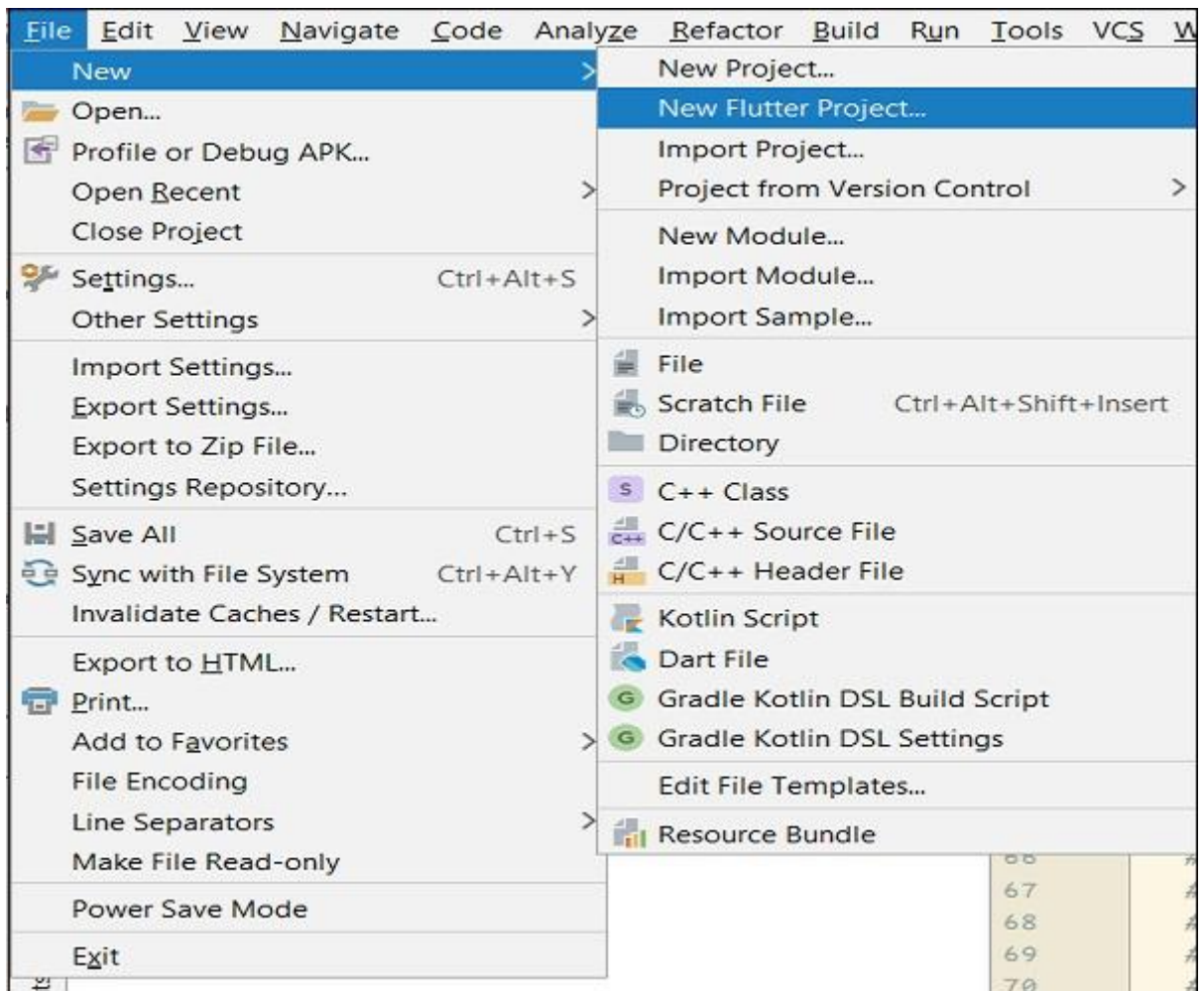
- Abra Android Studio
- Haga clic en **Preferencias** → **Complementos**
- Seleccione el complemento Flutter y haga clic en Instalar
- Haga clic en Sí cuando se le solicite que instale el complemento Dart.
- Reinicia el estudio de Android.

Crear una aplicación simple en Android Studio

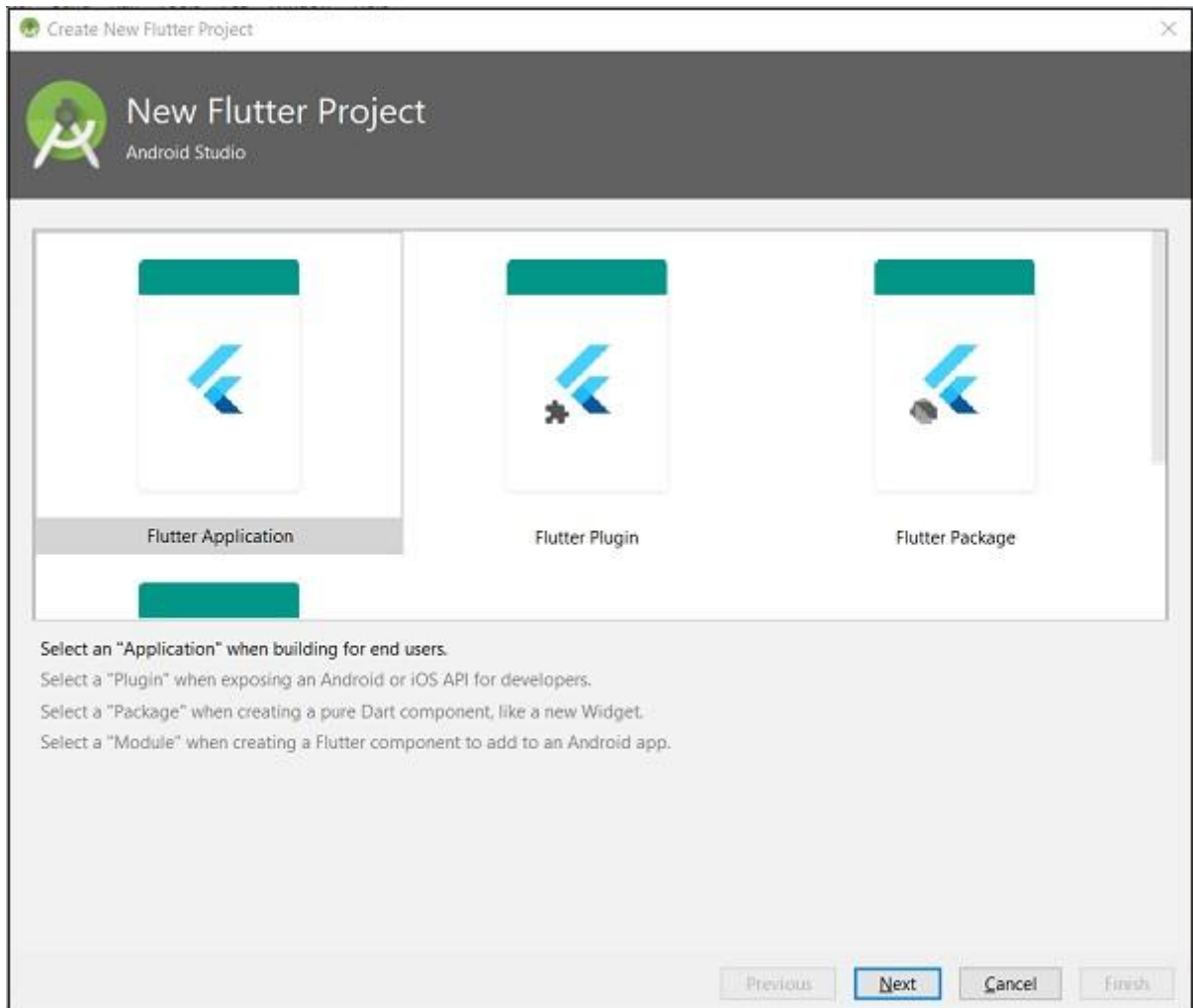
En este capítulo, *creemos una* aplicación *Flutter* simple para comprender los conceptos básicos de la creación de una aplicación Flutter en Android Studio.

Paso 1 - Abra Android Studio

Paso 2 : crea el proyecto Flutter. Para esto, haga clic en **Archivo** → **Nuevo** → **Nuevo proyecto Flutter**

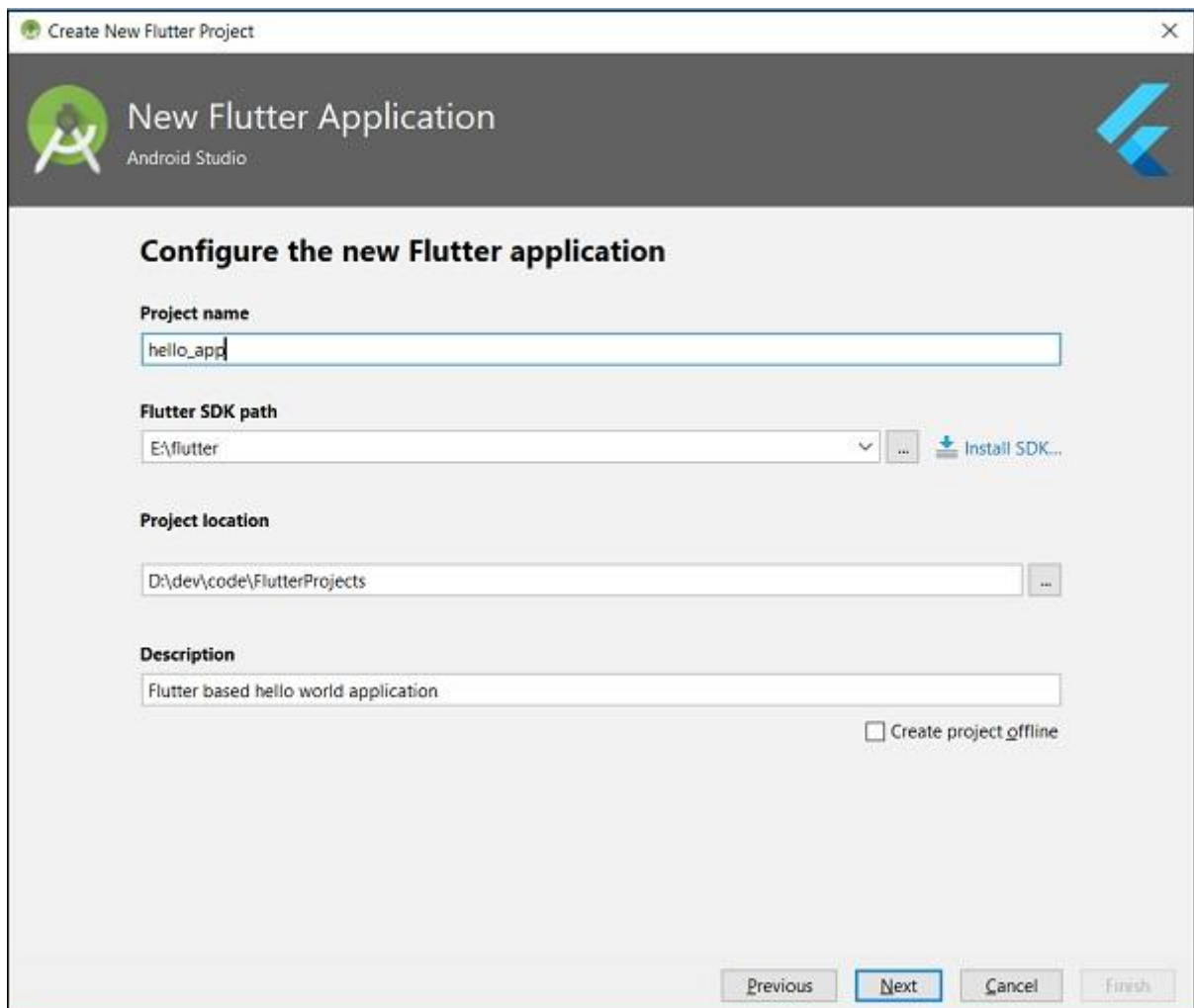


Paso 3 : selecciona la aplicación Flutter. Para esto, seleccione **Aplicación Flutter** y haga clic en **Siguiente** .



Paso 4 : configure la aplicación como se muestra a continuación y haga clic en **Siguiente** .

- Nombre del proyecto: **hello_app**
- Flutter SDK Path: **<path_to_flutter_sdk>**
- Ubicación del proyecto: **<path_to_project_folder>**
- Descripción: **aplicación hola mundo basada en Flutter**



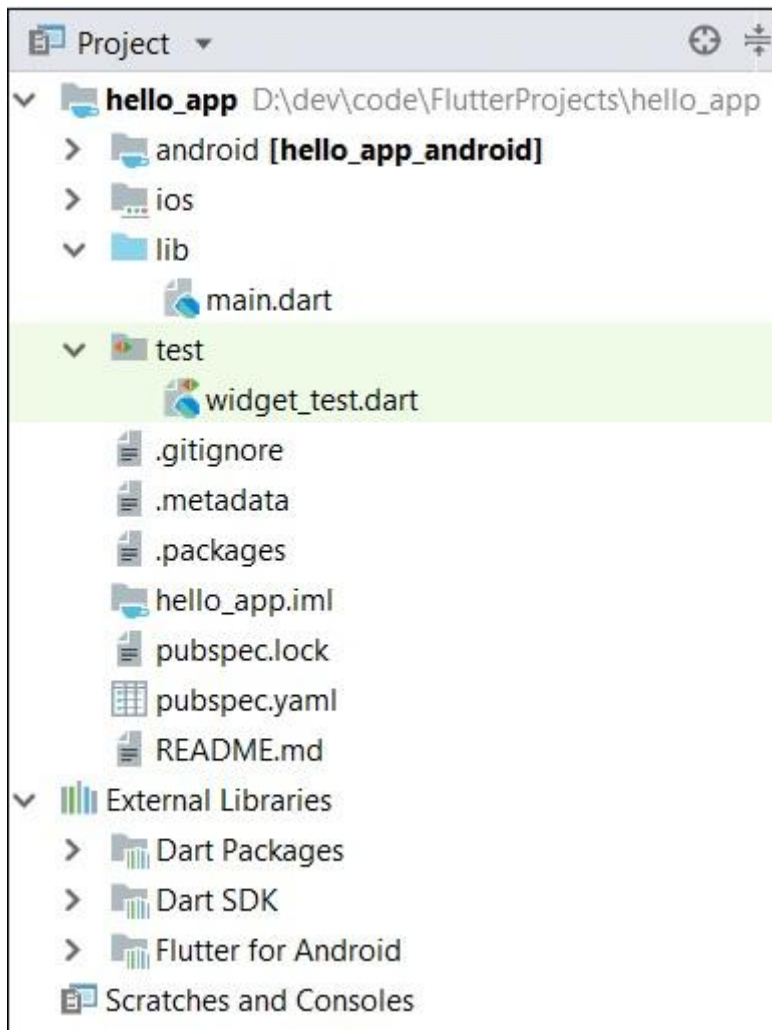
Paso 5 : configurar el proyecto.

Establezca el dominio de la empresa como **flutterapp.postparaprogramadores.com** y haga clic en **Finalizar** .

Paso 6 : ingrese el dominio de la empresa.

Android Studio crea una aplicación de Flutter totalmente funcional con una funcionalidad mínima. Verifiquemos la estructura de la aplicación y luego, cambiemos el código para realizar nuestra tarea.

La estructura de la aplicación y su finalidad es la siguiente:



Aquí se explican varios componentes de la estructura de la aplicación:

- **android** : código fuente generado automáticamente para crear una aplicación de Android
- **ios** : código fuente generado automáticamente para crear la aplicación ios
- **lib** - Carpeta principal que contiene el código Dart escrito usando el marco flutter
- **lib / main.dart** : punto de entrada de la aplicación Flutter
- **test** - Carpeta que contiene el código Dart para probar la aplicación flutter
- **test / widget_test.dart** - Código de muestra
- **.gitignore** - Archivo de control de versiones de Git
- **.metadata** : generada automáticamente por las herramientas flutter
- **.packages** : generado automáticamente para rastrear los paquetes de flutter
- **.iml** : archivo de proyecto utilizado por Android studio
- **pubspec.yaml** : utilizado por **Pub** , administrador de paquetes Flutter
- **pubspec.lock** : generado automáticamente por el administrador de paquetes Flutter, **Pub**
- **README.md** : archivo de descripción del proyecto escrito en formato Markdown

Paso 7: reemplace el código de dardo en el archivo *lib / main.dart* con el siguiente código:

```
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: 'Hello World Demo Application',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Home page'),
    );
  }
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: Text(
          'Hello World',
        ),
      ),
    );
  }
}
```

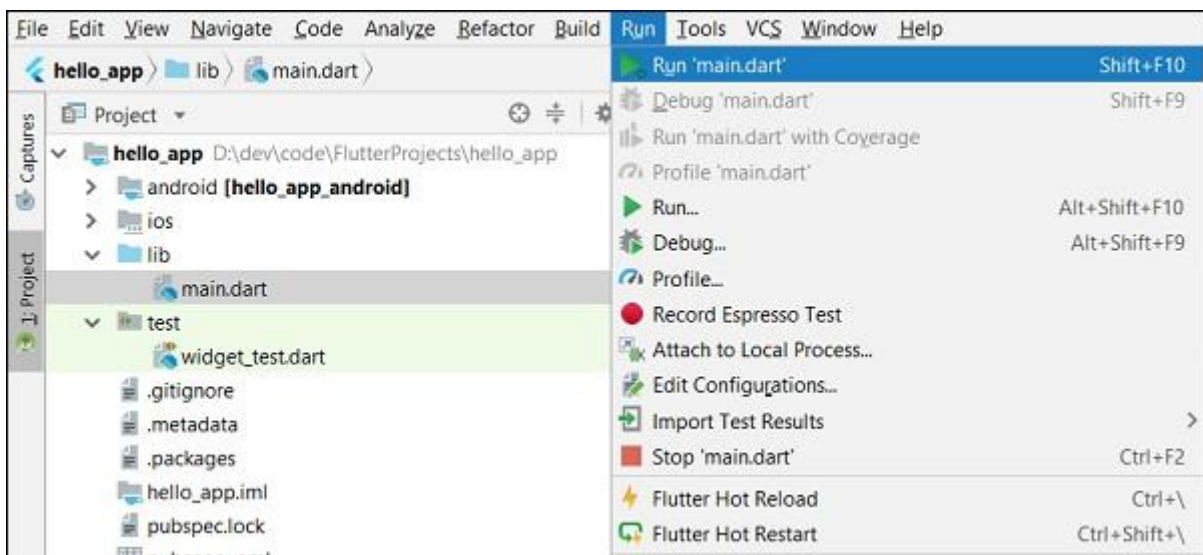
Vamos a entender el código de dardo línea por línea.

- **Línea 1:** importa el paquete de Flutter, *material*. El material es un paquete de Flutter para crear una interfaz de usuario de acuerdo con las pautas de diseño de material especificadas por Android.
- **Línea 3:** este es el punto de entrada de la aplicación Flutter. Llama a la función *runApp* y le pasa un objeto de la clase *MyApp*. El propósito de la función *runApp* es adjuntar el widget dado a la pantalla.
- **Línea 5-17:** el widget se usa para crear la interfaz de usuario en el marco de flutter. *StatelessWidget* es un widget que no mantiene ningún estado del widget. *MyApp* extiende *StatelessWidget* y anula su *método de compilación*. El propósito del método de *compilación* es crear una parte de la interfaz de usuario de la aplicación. Aquí, el método de *compilación* utiliza *MaterialApp*, un widget

para crear la interfaz de usuario de nivel raíz de la aplicación. Tiene tres propiedades: *título*, *tema* y *hogar*.

- *título* es el título de la aplicación
- *tema* es el tema del widget. Aquí, configuramos el *azul* como el color general de la aplicación usando la clase *ThemeData* y su propiedad, *primarySwatch*.
- *home* es la interfaz de usuario interna de la aplicación, que configuramos otro widget, **MyHomePage**
- **Línea 19 - 38** - *MyHomePage* es igual que *MyApp* excepto que devuelve *Scaffold* Widget. *Scaffold* es un widget de nivel superior junto al widget de *MaterialApp* que se utiliza para crear un diseño de material conforme a la interfaz de usuario. Tiene dos propiedades importantes, *appBar* para mostrar el encabezado de la aplicación y el cuerpo para mostrar el contenido real de la aplicación. *AppBar* es otro widget para representar el encabezado de la aplicación y lo hemos usado en la propiedad *appBar*. En la propiedad del *cuerpo*, hemos utilizado el widget *Center*, que lo centra como widget secundario. *Texto* es el widget final y más interno para mostrar el texto y se muestra en el centro de la pantalla.

Paso 8 - Ahora, ejecuta la aplicación usando, **Ejecutar** → **Ejecutar main.dart**



Paso 9 - Finalmente, el resultado de la aplicación es el siguiente:



Flutter - Aplicación de arquitectura

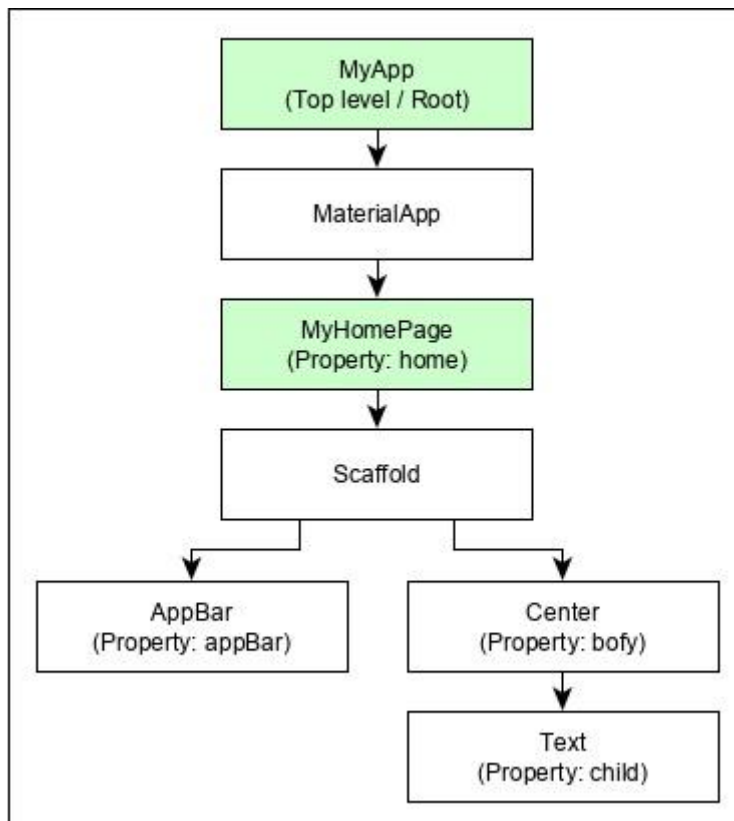
En este capítulo, analicemos la arquitectura del marco Flutter.

Widgets

El concepto central del marco de Flutter es **In Flutter, Everything es un widget**. Los widgets son básicamente componentes de la interfaz de usuario utilizados para crear la interfaz de usuario de la aplicación.

En *Flutter*, la aplicación es en sí misma un widget. La aplicación es el widget de nivel superior y su interfaz de usuario se crea utilizando uno o más elementos secundarios (widgets), que nuevamente se crean utilizando sus elementos secundarios. Esta característica de **componibilidad** nos ayuda a crear una interfaz de usuario de cualquier complejidad.

Por ejemplo, la jerarquía de widgets de la aplicación hello world (creada en el capítulo anterior) es como se especifica en el siguiente diagrama:



Aquí vale la pena destacar los siguientes puntos:

- *MyApp* es el widget creado por el usuario y se crea utilizando el widget nativo de Flutter, *MaterialApp*.
- *MaterialApp* tiene una propiedad de inicio para especificar la interfaz de usuario de la página de inicio, que nuevamente es un widget creado por el usuario, *MyHomePage*.
- *MyHomePage* se construye utilizando otro widget nativo de flutter, *Scaffold*
- *El andamio* tiene dos propiedades: *cuerpo* y *appBar*
- *body* se usa para especificar su interfaz de usuario principal y *appBar* se usa para especificar su interfaz de usuario de encabezado
- *La interfaz de usuario de encabezado* se crea utilizando el widget nativo de flutter, la *interfaz de usuario de AppBar* y *Body* se crea utilizando el widget de *Center*
- El widget *Center* tiene una propiedad, *Child*, que hace referencia al contenido real y se crea utilizando el widget *Text*

Gestos

Los widgets Flutter admiten la interacción a través de un widget especial, *GestureDetector*. *GestureDetector* es un widget invisible que tiene la capacidad de capturar interacciones del usuario, como tocar, arrastrar, etc., de su widget secundario. Muchos widgets nativos de Flutter admiten la interacción mediante el uso de *GestureDetector*. También podemos incorporar características interactivas en el widget existente al componerlo con el widget *GestureDetector*. Aprenderemos los gestos por separado en los próximos capítulos.

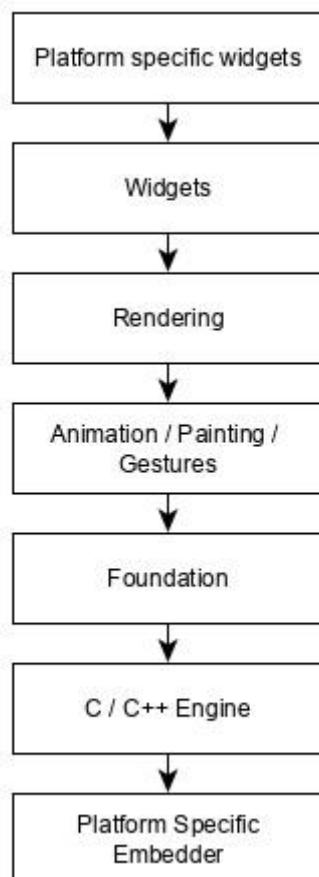
Concepto de Estado

Los widgets Flutter admiten el *mantenimiento del estado* al proporcionar un widget especial, *StatefulWidget*. El widget debe derivarse del widget *StatefulWidget* para admitir el mantenimiento del estado y todos los demás widgets deben derivarse de *StatefulWidget*. Los widgets de Flutter son **reactivos** en nativo. Esto es similar a reactjs y *StatefulWidget* se auto re-vuelve cada vez que se cambia su estado interno. La nueva representación se optimiza al encontrar la diferencia entre la interfaz de usuario de widgets antigua y nueva y mostrar solo los cambios necesarios

Capas

El concepto más importante del marco Flutter es que el marco está agrupado en múltiples categorías en términos de complejidad y claramente organizado en capas de complejidad decreciente. Una capa se construye utilizando su capa inmediata de nivel siguiente. La capa superior es un widget específico para *Android* e *iOS*. La siguiente capa tiene todos los widgets nativos de Flutter. La siguiente capa es la capa de *renderizado*, que es un componente de renderizador de bajo nivel y renderiza todo en la aplicación flutter. Las capas bajan al código específico de la plataforma

La descripción general de una capa en Flutter se especifica en el siguiente diagrama:



Los siguientes puntos resumen la arquitectura de Flutter:

- En Flutter, todo es un widget y un widget complejo está compuesto por widgets ya existentes.
- Las características interactivas se pueden incorporar cuando sea necesario utilizando el widget *GestureDetector*.
- El estado de un widget se puede mantener cuando sea necesario utilizando el widget *StatefulWidget*.
- Flutter ofrece un diseño en capas para que cualquier capa pueda ser programada dependiendo de la complejidad de la tarea.

Discutiremos todos estos conceptos en detalle en los próximos capítulos.

Flutter - Introducción a la programación de dardos

Dart es un lenguaje de programación de uso general de código abierto. Fue desarrollado originalmente por Google. Dart es un lenguaje orientado a objetos con sintaxis de estilo C. Admite conceptos de programación como interfaces, clases, a diferencia de otros lenguajes de programación Dart no admite matrices. Las colecciones de Dart se pueden usar para replicar estructuras de datos como matrices, genéricos y tipo opcional.

El siguiente código muestra un simple programa Dart:

```
void main() {  
    print("Dart language is easy to learn");  
}
```

Variables y tipos de datos

La *variable* se denomina ubicación de almacenamiento y los *tipos de datos* simplemente se refieren al tipo y tamaño de datos asociados con variables y funciones.

Dart usa la palabra clave *var* para declarar la variable. La sintaxis de *var* se define a continuación,

```
var name = 'Dart';
```

Las palabras clave *final* y *const* se usan para declarar constantes. Se definen a continuación:

```
void main() {  
    final a = 12;  
    const pi = 3.14;  
    print(a);  
    print(pi);  
}
```

El lenguaje Dart admite los siguientes tipos de datos:

- **Números** : se utiliza para representar literales numéricos: entero y doble.

- **Cadenas** : representa una secuencia de caracteres. Los valores de cadena se especifican en comillas simples o dobles.
- **Booleanos** : Dart utiliza la palabra clave *bool* para representar valores booleanos: verdadero y falso.
- **Listas y mapas** : se utiliza para representar una colección de objetos. Una lista simple se puede definir como a continuación -.

```
void main() {
  var list = [1,2,3,4,5];
  print(list);
}
```

La lista que se muestra arriba produce la lista [1,2,3,4,5].

El mapa se puede definir como se muestra aquí:

```
void main() {
  var mapping = {'id': 1, 'name': 'Dart'};
  print(mapping);
}
```

- **Dinámico** : si el tipo de variable no está definido, su tipo predeterminado es dinámico. El siguiente ejemplo ilustra la variable de tipo dinámico:

```
void main() {
  dynamic name = "Dart";
  print(name);
}
```

Toma de decisiones y bucles

Un bloque de toma de decisiones evalúa una condición antes de que se ejecuten las instrucciones. Dart admite If, If..else y declaraciones de cambio.

Los bucles se utilizan para repetir un bloque de código hasta que se cumpla una condición específica. Dart admite bucles for, for..in, while y do.. while.

Comprendamos un ejemplo simple sobre el uso de sentencias de control y bucles:

```
void main() {
  for( var i = 1 ; i <= 10; i++ ) {
    if(i%2==0) {
      print(i);
    }
  }
}
```

El código anterior imprime los números pares del 1 al 10.

Las funciones

Una función es un grupo de declaraciones que juntas realizan una tarea específica. Veamos una función simple en Dart como se muestra aquí:

```

void main() {
    add(3,4);
}
void add(int a,int b) {
    int c;
    c = a+b;
    print(c);
}

```

La función anterior agrega dos valores y produce 7 como salida.

Programación orientada a objetos

Dart es un lenguaje orientado a objetos. Admite funciones de programación orientada a objetos como clases, interfaces, etc.

Una clase es un plan para crear objetos. Una definición de clase incluye lo siguiente:

- Campos
- Getters y setters
- Constructores
- Las funciones

Ahora, creemos una clase simple usando las definiciones anteriores:

```

class Employee {
    String name;

    //getter method
    String get emp_name {
        return name;
    }
    //setter method
    void set emp_name(String name) {
        this.name = name;
    }
    //function definition
    void result() {
        print(name);
    }
}

void main() {
    //object creation
    Employee emp = new Employee();
    emp.name = "employee1";
    emp.result(); //function call
}

```

Flutter - Introducción a los widgets

Como aprendimos en el capítulo anterior, los widgets son todo en el marco de Flutter. Ya hemos aprendido cómo crear nuevos widgets en capítulos anteriores.

En este capítulo, comprendamos el concepto real detrás de la creación de widgets y los diferentes tipos de widgets disponibles en el marco de *Flutter*.

Revisemos el widget *MyHomePage* de la aplicación *Hello World*. El código para este propósito es el siguiente:

```
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(this.title), ),
      body: Center(child: Text( 'Hello World',)),
    );
  }
}
```

Aquí, hemos creado un nuevo widget al extender *StatelessWidget*.

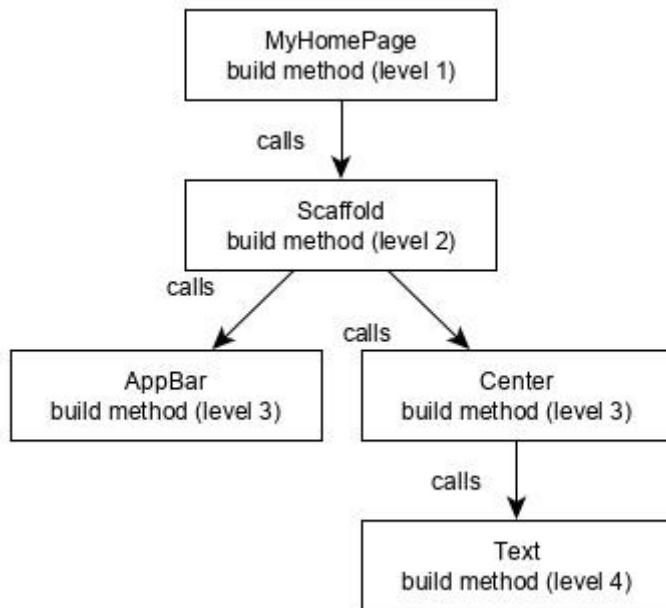
Tenga en cuenta que *StatelessWidget* solo requiere que se implemente una *compilación de método* único en su clase derivada. El método de *compilación* obtiene el entorno de contexto necesario para compilar los widgets a través del parámetro *BuildContext* y devuelve el widget que construye.

En el código, hemos usado *title* como uno de los argumentos del constructor y también hemos usado *Key* como otro argumento. El *título* se usa para mostrar el título y la *clave* se usa para identificar el widget en el entorno de compilación.

En este caso, la *acumulación* método llama a la *acumulación* método de *Andamios*, que a su vez llama a la *acumulación* método de *AppBar* y *Centro* de *construir* su interfaz de usuario.

Finalmente, el método de compilación *Center* llama al método de compilación de *texto*.

Para una mejor comprensión, la representación visual de la misma se da a continuación:



Visualización de compilación de widgets

En *Flutter*, los widgets se pueden agrupar en varias categorías según sus características, como se detalla a continuación:

- Widgets específicos de plataforma
- Widgets de diseño
- Widgets de mantenimiento del estado
- Plataforma independiente / widgets básicos

Discutamos cada uno de ellos en detalle ahora.

Widgets específicos de plataforma

Flutter tiene widgets específicos para una plataforma en particular: Android o iOS.

Los widgets específicos de Android están diseñados de acuerdo con *las pautas de diseño de materiales* del sistema operativo Android. Los widgets específicos de Android se denominan *widgets de material*.

Los widgets específicos de iOS están diseñados de acuerdo con *las pautas de interfaz humana* de Apple y se denominan widgets de *Cupertino*.

Algunos de los widgets de material más utilizados son los siguientes:

- Andamio
- AppBar
- FondoNavegaciónBar
- Barra de pestañas
- TabBarView
- ListTile

- Botón elevado
- FlotanteAcciónBotón
- FlatButton
- IconButton
- Botón desplegable
- PopupMenuButton
- ButtonBar
- Campo de texto
- Caja
- Radio
- Cambiar
- Control deslizante
- Selectores de fecha y hora
- SimpleDialog
- AlertDialog

Algunos de los widgets de *Cupertino* más utilizados son los siguientes:

- CupertinoButton
- CupertinoPicker
- CupertinoDatePicker
- CupertinoTimerPicker
- CupertinoNavegaciónBar
- CupertinoTabBar
- CupertinoTabAndamio
- CupertinoTabView
- CupertinoTextField
- CupertinoDialog
- CupertinoDialogAction
- CupertinoFullscreenDialogTransition
- CupertinoPáginaAndamio
- CupertinoPageTransition
- CupertinoActionSheet
- CupertinoActividadIndicador
- CupertinoAlertDialog
- CupertinoPopupSurface
- CupertinoSlider

Widgets de diseño

En Flutter, se puede crear un widget componiendo uno o más widgets. Para componer varios widgets en un solo widget, *Flutter* proporciona una gran cantidad de widgets con la función de diseño. Por ejemplo, el widget secundario se puede centrar utilizando el widget *Center*.

Algunos de los widgets de diseño populares son los siguientes:

- **Contenedor** : una caja rectangular decorada con widgets de *BoxDecoration* con fondo, borde y sombra.
- **Centrar** : centra su widget secundario.
- **Fila** : organice sus elementos secundarios en la dirección horizontal.
- **Columna** : organice sus elementos secundarios en la dirección vertical.
- **Apilar** : organizar uno encima del otro.

Comprobaremos los widgets de diseño en detalle en el próximo capítulo *Introducción a los widgets de diseño* .

Widgets de mantenimiento del estado

En Flutter, todos los widgets se derivan de *StatelessWidget* o *StatefulWidget* .

El widget derivado de *StatelessWidget* no tiene ninguna información de estado, pero puede contener un widget derivado de *StatefulWidget* . La naturaleza dinámica de la aplicación es a través del comportamiento interactivo de los widgets y los cambios de estado durante la interacción. Por ejemplo, al tocar un botón de contador aumentará / disminuirá el estado interno del contador en uno y la naturaleza reactiva del widget *Flutter* volverá a representar el widget automáticamente con nueva información de estado.

Aprenderemos el concepto de widgets de *StatefulWidget* en detalle en el próximo capítulo *de gestión de estado* .

Plataforma independiente / widgets básicos

Flutter proporciona una gran cantidad de widgets básicos para crear una interfaz de usuario simple y compleja de manera independiente de la plataforma. Veamos algunos de los widgets básicos en este capítulo.

Texto

El widget de *texto* se usa para mostrar un trozo de cadena. El estilo de la cadena se puede establecer utilizando la propiedad de *estilo* y la clase *TextStyle* . El código de muestra para este propósito es el siguiente:

```
Text('Hello World!', style: TextStyle(fontWeight:
FontWeight.bold))
```

El widget de *texto* tiene un constructor especial, *Text.rich* , que acepta el elemento secundario de tipo *TextSpan* para especificar la cadena con un estilo diferente. El widget *TextSpan* es de naturaleza recursiva y acepta *TextSpan* como sus hijos. El código de muestra para este propósito es el siguiente:

```
Text.rich(
  TextSpan(
    children: <TextSpan>[
      TextSpan(text: "Hello ", style:
        TextStyle(fontStyle: FontStyle.italic)),
      TextSpan(text: "World", style:
        TextStyle(fontWeight: FontWeight.bold)),
```

```
    ],  
  ),  
)
```

Las propiedades más importantes del widget de *texto* son las siguientes:

- **maxLines, int** - Número máximo de líneas para mostrar
- **overflow, TextOverflow** : especifique cómo se maneja el desbordamiento visual usando la clase *TextOverflow*
- **style, TextStyle** : especifique el estilo de la cadena usando la clase *TextStyle*
- **textAlign, TextAlign** : alineación del texto como derecha, izquierda, justificar, etc., usando la clase *TextAlign*
- **textDirection, TextDirection** - Dirección del texto a fluir, de izquierda a derecha o de derecha a izquierda

Imagen

El widget de *imagen* se usa para mostrar una imagen en la aplicación. El widget de *imagen* proporciona diferentes constructores para cargar imágenes de múltiples fuentes y son los siguientes:

- **Image** : cargador de imágenes genérico con *ImageProvider*
- **Image.asset** : carga la imagen de los activos del proyecto flutter
- **Image.file** : carga la imagen de la carpeta del sistema
- **Image.memory** - Cargar imagen desde la memoria
- **Image.Network** - Cargar imagen desde la red

La opción más fácil para cargar y mostrar una imagen en *Flutter* es incluir la imagen como activos de la aplicación y cargarla en el widget a pedido.

- Cree una carpeta, activos en la carpeta del proyecto y coloque las imágenes necesarias.
- Especifique los activos en pubspec.yaml como se muestra a continuación:

```
flutter:  
  assets:  
    - assets/smiley.png
```

- Ahora, cargue y muestre la imagen en la aplicación.

```
Image.asset('assets/smiley.png')
```

- El código fuente completo del widget *MyHomePage* de la aplicación hello world y el resultado es el que se muestra a continuación.

```
class MyHomePage extends StatelessWidget {  
  MyHomePage({Key key, this.title}) : super(key: key);  
  final String title;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar( title: Text(this.title), ),  
    );  
  }  
}
```



```
body: Center( child:
Image.asset("assets/smiley.png")),
);
}
}
```

La imagen cargada se muestra a continuación:



Las propiedades más importantes del widget *Imagen* son las siguientes:

- **image, ImageProvider** - Imagen real para cargar
- **ancho, doble** - Ancho de la imagen
- **altura, doble** - Altura de la imagen
- **alineación, Alineación Geometría** : cómo alinear la imagen dentro de sus límites

Icono

El widget *Icon* se usa para mostrar un glifo de una fuente descrita en la clase *IconData*. El código para cargar un ícono de correo electrónico simple es el siguiente:

```
Icon(Icons.email)
```

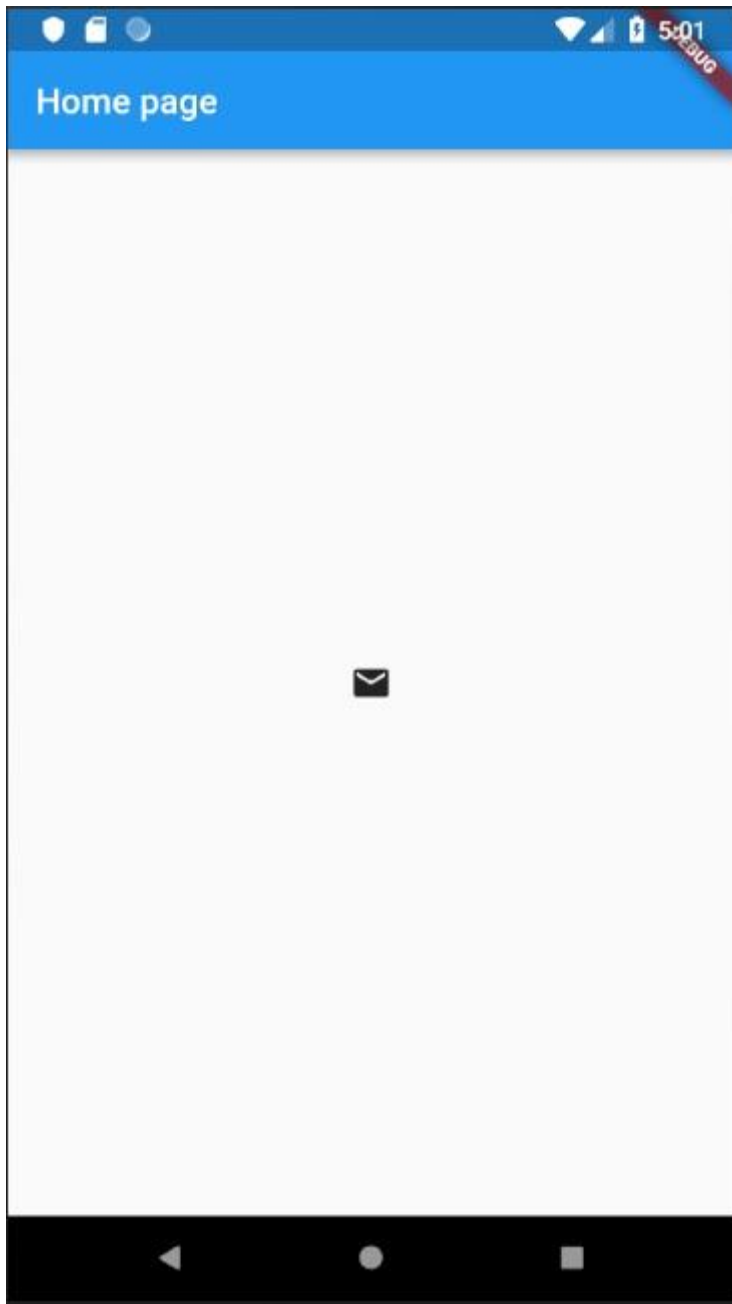
El código fuente completo para aplicarlo en la aplicación hello world es el siguiente:

```
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
```

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(title: Text(this.title)),  
    body: Center( child: Icon(Icons.email)),  
  );  
}
```

El ícono cargado se muestra a continuación:



Flutter - Introducción a los diseños

Dado que el concepto central de *Flutter* es *Todo es widget*, *Flutter* incorpora una funcionalidad de diseño de interfaz de usuario en los propios widgets. *Flutter* proporciona bastantes widgets especialmente diseñados como *Contenedor*, *Centro*, *Alinear*, etc., solo con el propósito de diseñar la interfaz de usuario. Los widgets contruidos componiendo otros widgets normalmente usan widgets de diseño. Aprendamos a usar el concepto de diseño de *Flutter* en este capítulo.

Tipo de widgets de diseño

Los widgets de diseño se pueden agrupar en dos categorías distintas según su elemento secundario:

- Widget que apoya a un niño soltero
- Widget que admite varios hijos

Aprendamos tanto el tipo de widgets como su funcionalidad en las próximas secciones.

Widgets para niños

En esta categoría, los widgets tendrán solo un widget como elemento secundario y cada widget tendrá una funcionalidad de diseño especial.

Por ejemplo, *Center* widget solo centra su widget secundario con respecto a su widget principal y el widget de *Contenedor* proporciona flexibilidad completa para colocarlo secundario en cualquier lugar dentro de él utilizando diferentes opciones como relleno, decoración, etc.

Los widgets secundarios individuales son excelentes opciones para crear widgets de alta calidad con una funcionalidad única, como botones, etiquetas, etc.

El código para crear un botón simple usando el widget de *Contenedor* es el siguiente:

```
class MyButton extends StatelessWidget {
  MyButton({Key key}) : super(key: key);

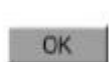
  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: const BoxDecoration(
        border: Border(
          top: BorderSide(width: 1.0, color:
Color(0xFFFFFFFF)),
          left: BorderSide(width: 1.0, color:
Color(0xFFFFFFFF)),
          right: BorderSide(width: 1.0, color:
Color(0xFFFF0000)),
```

```

        bottom: BorderSide(width: 1.0, color:
Color(0xFFFF000000)),
    ),
    ),
    child: Container(
        padding: const
        EdgeInsets.symmetric(horizontal: 20.0, vertical:
2.0),
        decoration: const BoxDecoration(
            border: Border(
                top: BorderSide(width: 1.0, color:
Color(0xFFFFDFDFDF)),
                left: BorderSide(width: 1.0, color:
Color(0xFFFFDFDFDF)),
                right: BorderSide(width: 1.0, color:
Color(0xFFFF7F7F7F)),
                bottom: BorderSide(width: 1.0, color:
Color(0xFFFF7F7F7F)),
            ),
            color: Colors.grey,
        ),
        child: const Text(
            'OK',textAlign: TextAlign.center, style:
TextStyle(color: Colors.black)
        ),
    ),
);
}
}

```

Aquí, hemos utilizado dos widgets: un widget de *contenedor* y un widget de *texto*. El resultado del widget es un botón personalizado como se muestra a continuación:



Revisemos algunos de los widgets de diseño secundarios únicos más importantes proporcionados por *Flutter*:

- **Relleno**: se utiliza para organizar su widget secundario según el relleno proporcionado. Aquí, el relleno puede ser proporcionado por la clase *EdgeInsets*.
- **Alinear**: alinee su widget secundario dentro de sí mismo utilizando el valor de la propiedad de *alineación*. El valor de la propiedad de *alineación* puede ser proporcionado por la clase *FractionalOffset*. La clase *FractionalOffset* especifica los desplazamientos en términos de una distancia desde la parte superior izquierda.

Algunos de los posibles valores de las compensaciones son los siguientes:

- *FractionalOffset* (1.0, 0.0) representa la esquina superior derecha.
- *FractionalOffset* (0.0, 1.0) representa la parte inferior izquierda.

A continuación se muestra un código de muestra sobre las compensaciones:

```

Center(
  child: Container(
    height: 100.0,
    width: 100.0,
    color: Colors.yellow, child: Align(
      alignment: FractionalOffset(0.2, 0.6),
      child: Container( height: 40.0, width:
        40.0, color: Colors.red,
      ),
    ),
  ),
),
)

```

- **FittedBox** : escala el widget secundario y luego lo coloca de acuerdo con el ajuste especificado.
- **AspectRatio** : intenta **ajustar el** tamaño del widget secundario a la relación de aspecto especificada
- Caja restringida
- Base
- FractionallySizedBox
- Altura intrínseca
- Ancho intrínseco
- LimitedBox
- Entre bastidores
- OverflowBox
- SizedBox
- SizedOverflowBox
- Transformar
- CustomSingleChildLayout

Nuestra aplicación hello world está utilizando widgets de diseño basados en materiales para diseñar la página de inicio. Modifiquemos nuestra aplicación hello world para construir la página de inicio utilizando widgets de diseño básicos como se especifica a continuación:

- **Contenedor** : widget de contenedor genérico, hijo único, basado en cajas con alineación, relleno, borde y margen junto con características de estilo enriquecidas.
- **Centro** : widget de contenedor secundario simple y único, que centra su widget secundario.

El código modificado del widget *MyHomePage* y *MyApp* es el siguiente:

```

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MyHomePage(title: "Hello World demo app");
  }
}

```

```

}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(color: Colors.white,),
      padding: EdgeInsets.all(25), child: Center(
        child: Text(
          'Hello World', style: TextStyle(
            color: Colors.black, letterSpacing: 0.5,
fontSize: 20,
          ),
          textDirection: TextDirection.ltr,
        ),
      ),
    );
  }
}

```

Aquí,

- El widget de *contenedor* es el widget de nivel superior o raíz. El *contenedor* se configura utilizando la propiedad de *decoración* y *relleno* para diseñar su contenido.
- *BoxDecoration* tiene muchas propiedades como color, borde, etc., para decorar el widget *Contenedor* y aquí, el *color* se usa para establecer el color del contenedor.
- El *relleno* del widget *Container* se establece mediante la clase *EdgeInsets*, que proporciona la opción de especificar el valor del relleno.
- *Centro* es el widget secundario del widget *Contenedor*. Nuevamente, *Text* es el hijo del widget *Center*. El *texto* se usa para mostrar el mensaje y el *Centro* se usa para centrar el mensaje de texto con respecto al widget principal, *Contenedor*.

El resultado final del código anterior es una muestra de diseño como se muestra a continuación:



Hello World



Múltiples widgets secundarios

En esta categoría, un widget dado tendrá más de un widget secundario y el diseño de cada widget es único.

Por ejemplo, el widget *Row* permite el diseño de sus elementos secundarios en dirección horizontal, mientras que el widget *Column* permite el diseño de sus elementos secundarios en dirección vertical. Al componer *Row* and *Column*, se puede construir un widget con cualquier nivel de complejidad.

Aprendamos algunos de los widgets de uso frecuente en esta sección.

- **Fila** : permite organizar sus elementos secundarios de manera horizontal.
- **Columna** : permite organizar sus elementos secundarios de forma vertical.
- **ListView** : permite organizar sus elementos secundarios como una lista.
- **GridView** : permite organizar sus elementos secundarios como galería.
- **Expandido** : se utiliza para hacer que los elementos secundarios del widget Fila y Columna ocupen el área máxima posible.

- **Tabla** : widget basado en tabla.
- **Flow** : widget basado en flujo.
- **Pila** : widget basado en pila.

Aplicación de diseño avanzado

En esta sección, aprendamos cómo crear una interfaz de usuario compleja de *listado de productos* con diseño personalizado utilizando widgets de diseño secundarios únicos y múltiples.

Para este propósito, siga la secuencia dada a continuación:

- Cree una nueva aplicación *Flutter* en el estudio de Android, *product_layout_app*.
- Reemplace el código *main.dart* con el siguiente código:

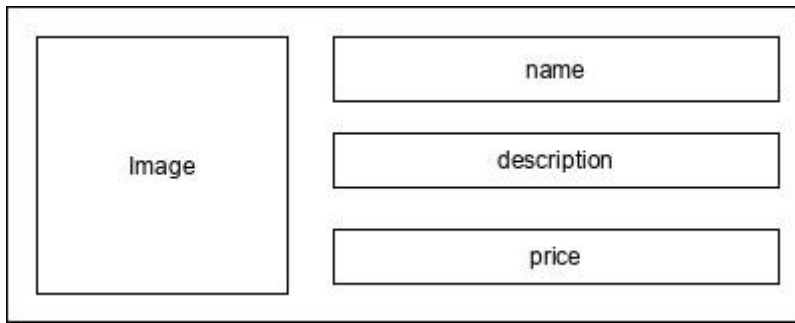
```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo', theme: ThemeData(
        primarySwatch: Colors.blue,
        home: MyHomePage(title: 'Product layout demo home
page'),
      );
  }
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(this.title)),
      body: Center(child: Text( 'Hello World', )),
    );
  }
}
```

- Aquí,
- Hemos creado *MyHomePage* Reproductor extendiendo *StatelessWidget* en lugar de por defecto *StatefulWidget* y luego se retira el código correspondiente.
- Ahora, cree un nuevo widget, *ProductBox* de acuerdo con el diseño especificado como se muestra a continuación:

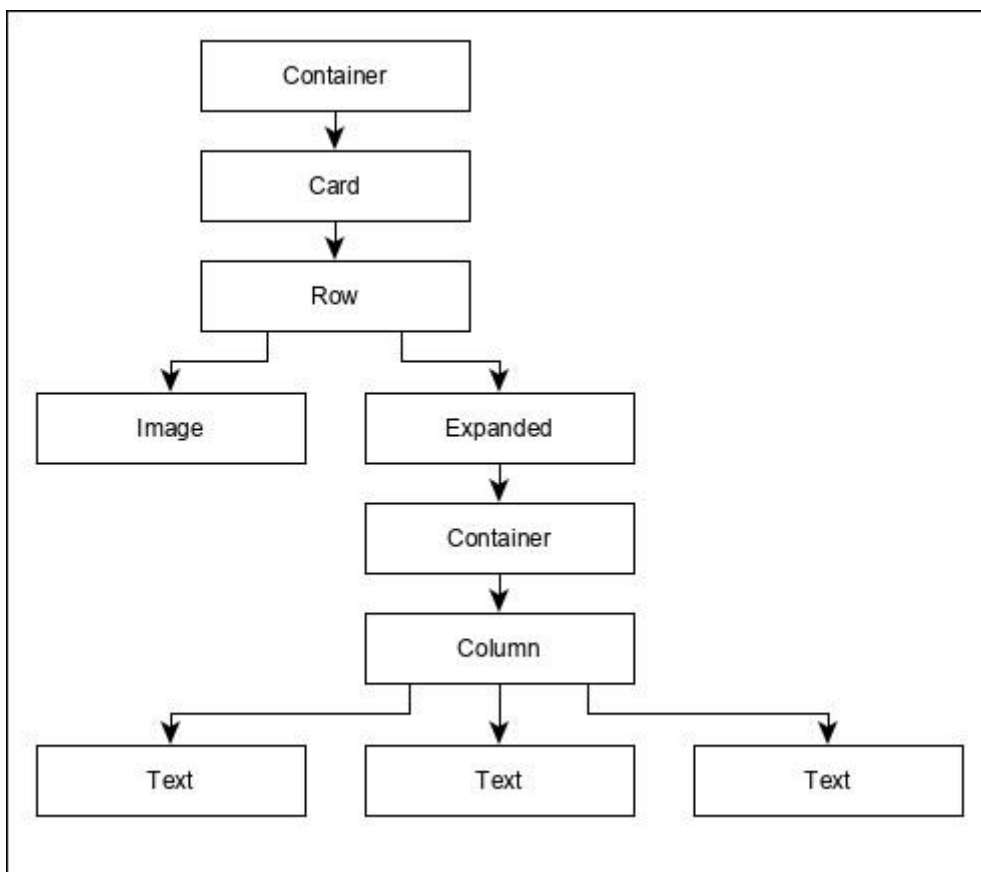


- El código para *ProductBox* es el siguiente.

```
class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.name, this.description,
    this.price, this.image})
    : super(key: key);
  final String name;
  final String description;
  final int price;
  final String image;

  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2), height: 120, child:
Card(
      child: Row(
        mainAxisAlignment:
MainAxisAlignment.spaceEvenly, children: <Widget>[
        Image.asset("assets/appimages/" +image),
Expanded(
          child: Container(
            padding: EdgeInsets.all(5), child:
Column(
              mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
              children: <Widget>[
                Text(this.name, style:
TextStyle(fontWeight:
                  FontWeight.bold)),
                Text(this.description),
                Text("Price: " +
this.price.toString()),
              ],
            ),
          ),
        ),
      ],
    ),
  );
}
```

- Tenga en cuenta lo siguiente en el código:
- *ProductBox* ha utilizado cuatro argumentos como se especifica a continuación:
 - nombre - Nombre del producto
 - description - Descripción del producto
 - precio - Precio del producto
 - imagen - Imagen del producto
- *ProductBox* utiliza siete widgets *integrados* como se especifica a continuación:
 - Envase
 - Expandido
 - Fila
 - Columna
 - Tarjeta
 - Texto
 - Imagen
- *ProductBox* está diseñado utilizando el widget mencionado anteriormente. La disposición o jerarquía del widget se especifica en el diagrama que se muestra a continuación:



- Ahora, coloque alguna imagen ficticia (consulte a continuación) para obtener información del producto en la carpeta de activos de la aplicación y configure la carpeta de activos en el archivo pubspec.yaml como se muestra a continuación:

```

assets:
  - assets/appimages/floppy.png

```

- assets/appimages/iphone.png
- assets/appimages/laptop.png
- assets/appimages/pendrive.png
- assets/appimages/pixel.png
- assets/appimages/tablet.png



iPhone

iPhone.png



pixel 1

Pixel.png



laptop

Laptop.png



tablet

Tablet.png



pen drive

Pendrive.png



Floppy.png

Finalmente, use el widget *ProductBox* en el widget *MyHomePage* como se especifica a continuación:

```
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Product Listing")),
      body: ListView(
        shrinkWrap: true, padding: const
EdgeInsets.fromLTRB(2.0, 10.0, 2.0, 10.0),
        children: <Widget> [
          ProductBox(
            name: "iPhone",
            description: "iPhone is the stylist phone
ever",
            price: 1000,
            image: "iphone.png"
          ),
          ProductBox(
            name: "Pixel",
            description: "Pixel is the most featureful
phone ever",
            price: 800,
            image: "pixel.png"
          ),
          ProductBox(
            name: "Laptop",
            description: "Laptop is most productive
development tool",
            price: 2000,
            image: "laptop.png"
          ),
          ProductBox(
            name: "Tablet",
            description: "Tablet is the most useful
device ever for meeting",
            price: 1500,
            image: "tablet.png"
          ),
          ProductBox(
            name: "Pendrive",
```

```

        description: "Pendrive is useful storage
medium",
        price: 100,
        image: "pendrive.png"
    ),
    ProductBox(
        name: "Floppy Drive",
        description: "Floppy drive is useful rescue
storage medium",
        price: 20,
        image: "floppy.png"
    ),
  ],
)
);
}
}

```

- Aquí, hemos utilizado *ProductBox* como hijos del widget *ListView*.
- El código completo (*main.dart*) de la aplicación de diseño del producto (*product_layout_app*) es el siguiente:

```

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(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Product layout demo home
page'),
    );
  }
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Product Listing")),
      body: ListView(
        shrinkWrap: true,
        padding: const EdgeInsets.fromLTRB(2.0, 10.0,
2.0, 10.0),
        children: <Widget>[
          ProductBox(
            name: "iPhone",

```

```

        description: "iPhone is the stylist phone
ever",
        price: 1000,
        image: "iphone.png"
    ),
    ProductBox(
        name: "Pixel",
        description: "Pixel is the most featureful
phone ever",
        price: 800,
        image: "pixel.png"
    ),
    ProductBox(
        name: "Laptop",
        description: "Laptop is most productive
development tool",
        price: 2000,
        image: "laptop.png"
    ),
    ProductBox(
        name: "Tablet",
        description: "Tablet is the most useful
device ever for meeting",
        price: 1500,
        image: "tablet.png"
    ),
    ProductBox(
        name: "Pendrive",
        description: "Pendrive is useful storage
medium",
        price: 100,
        image: "pendrive.png"
    ),
    ProductBox(
        name: "Floppy Drive",
        description: "Floppy drive is useful rescue
storage medium",
        price: 20,
        image: "floppy.png"
    ),
],
)
);
}

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.name, this.description,
this.price, this.image}) :
    super(key: key);
  final String name;
  final String description;
  final int price;
  final String image;
}

```

```

Widget build(BuildContext context) {
  return Container(
    padding: EdgeInsets.all(2),
    height: 120,
    child: Card(
      child: Row(
        mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Image.asset("assets/appimages/" + image),
          Expanded(
            child: Container(
              padding: EdgeInsets.all(5),
              child: Column(
                mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                children: <Widget>[
                  Text(
                    this.name, style: TextStyle(
                      fontWeight:
FontWeight.bold
                    ),
                  ),
                  Text(this.description), Text(
                    "Price: " +
this.price.toString()
                  ),
                ],
              ),
            ),
          ),
        ],
      ),
    ),
  );
}

```

El resultado final de la aplicación es el siguiente:



Flutter - Introducción a los gestos

Los *gestos* son principalmente una forma para que un usuario interactúe con una aplicación móvil (o cualquier dispositivo táctil). Los gestos se definen generalmente como cualquier acción / movimiento físico de un usuario con la intención de activar un control específico del dispositivo móvil. Los gestos son tan simples como tocar la pantalla del dispositivo móvil para acciones más complejas utilizadas en aplicaciones de juegos.

Aquí se mencionan algunos de los gestos más utilizados:

- **Tocar** : tocar la superficie del dispositivo con la punta del dedo durante un período breve y luego soltar la punta del dedo.
- **Doble toque**: **toca** dos veces en poco tiempo.

- **Arrastrar** : tocar la superficie del dispositivo con la punta del dedo y luego mover la punta del dedo de manera constante y finalmente soltar la punta del dedo.
- **Flick** : similar a arrastrar, pero hacerlo de forma veloz.
- **Pellizcar** : pellizca la superficie del dispositivo con dos dedos.
- **Difusión / Zoom** : opuesto al pellizco.
- **Desplazamiento panorámico** : tocar la superficie del dispositivo con la punta del dedo y moverlo en cualquier dirección sin soltar la punta del dedo.

Flutter ofrece un excelente soporte para todo tipo de gestos a través de su widget exclusivo, **GestureDetector** . GestureDetector es un widget no visual utilizado principalmente para detectar el gesto del usuario. Para identificar un gesto dirigido a un widget, el widget se puede colocar dentro del widget GestureDetector. GestureDetector capturará el gesto y enviará múltiples eventos basados en el gesto.

Algunos de los gestos y los eventos correspondientes se dan a continuación:

- Grifo
 - onTapDown
 - onTapUp
 - onTap
 - onTapCancel
- Doble toque
 - onDoubleTap
- Pulsación larga
 - onLongPress
- Arrastre vertical
 - onVerticalDragStart
 - onVerticalDragUpdate
 - onVerticalDragEnd
- Arrastre horizontal
 - onHorizontalDragStart
 - onHorizontalDragUpdate
 - onHorizontalDragEnd
- Pan
 - onPanStart
 - onPanUpdate
 - onPanEnd

Ahora, modifiquemos la aplicación hello world para incluir la función de detección de gestos e intentemos entender el concepto.

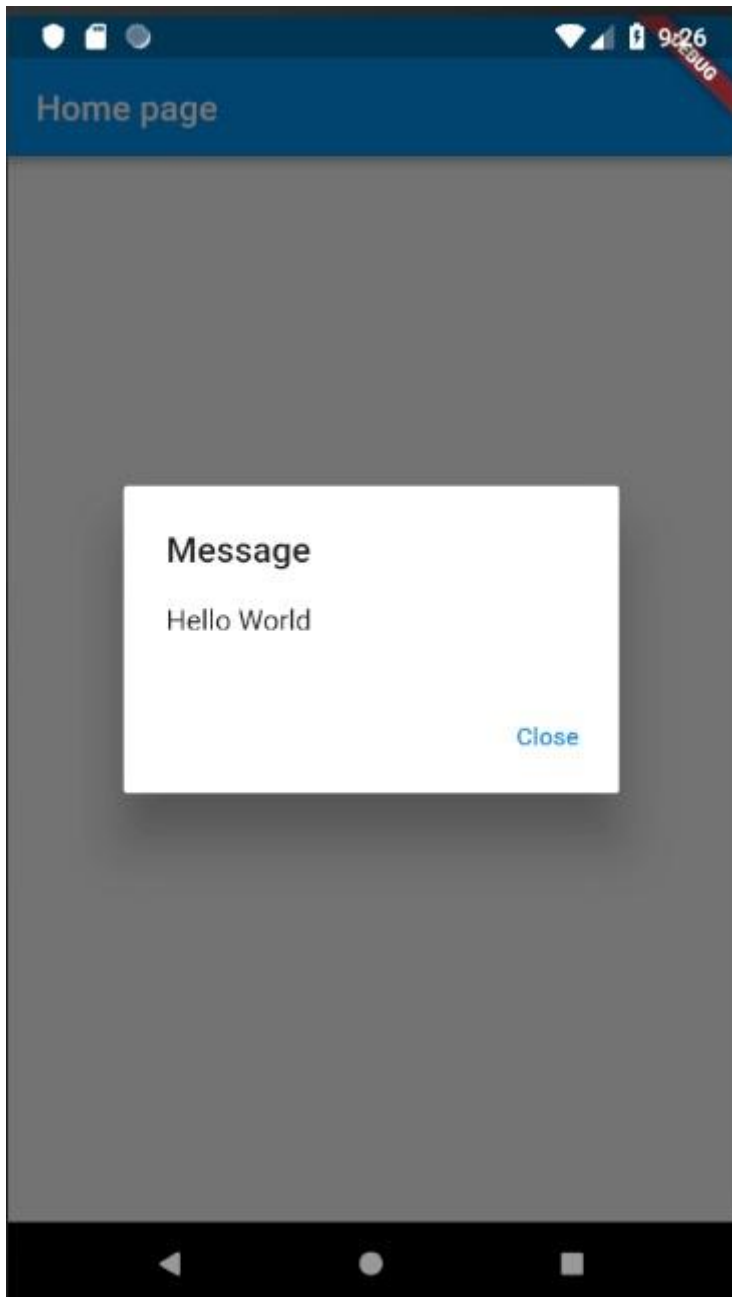
- Cambie el contenido del cuerpo del widget *MyHomePage* como se muestra a continuación:

```
body: Center(
  child: GestureDetector(
    onTap: () {
      _showDialog(context);
    },
    child: Text( 'Hello World', )
  )
),
```

- Observe que aquí hemos colocado el widget *GestureDetector* encima del widget *Text* en la jerarquía de widgets, capturamos el evento *onTap* y finalmente mostramos una ventana de diálogo.
- Implemente la función * *_showDialog* * para presentar un diálogo cuando el usuario toque el mensaje *hello world*. Utiliza el widget genérico *showDialog* y *AlertDialog* para crear un nuevo widget de diálogo. El código se muestra a continuación:

```
// user defined function void _showDialog(BuildContext
context) {
  // flutter defined function
  showDialog(
    context: context, builder: (BuildContext context) {
      // return object of type Dialog
      return AlertDialog(
        title: new Text("Message"),
        content: new Text("Hello World"),
        actions: <Widget>[
          new FlatButton(
            child: new Text("Close"),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      );
    },
  );
}
```

- La aplicación se volverá a cargar en el dispositivo utilizando la función de recarga en caliente. Ahora, simplemente haga clic en el mensaje *Hello World* y se mostrará el cuadro de diálogo a continuación:



- Ahora, cierre el cuadro de diálogo haciendo clic en la opción de *cierre* en el cuadro de diálogo.
- El código completo (main.dart) es el siguiente:

```
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: 'Hello World Demo Application',
      theme: ThemeData( primarySwatch: Colors.blue, ),
      home: MyHomePage(title: 'Home page'),
    );
  }
}
```

```

    );
  }
}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  // user defined function
  void _showDialog(BuildContext context) {
    // flutter defined function showDialog(
    context: context, builder: (BuildContext context) {
      // return object of type Dialog return
      AlertDialog(
        title: new Text("Message"),
        content: new Text("Hello World"),
        actions: <Widget>[
          new FlatButton(
            child: new Text("Close"),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      );
    },
  );
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text(this.title)),
    body: Center(
      child: GestureDetector(
        onTap: () {
          _showDialog(context);
        },
        child: Text( 'Hello World', )
      ),
    ),
  );
}
}

```

Finalmente, Flutter también proporciona un mecanismo de detección de gestos de bajo nivel a través del widget *Listener*. Detectará todas las interacciones del usuario y luego enviará los siguientes eventos:

- PointerDownEvent
- PointerMoveEvent
- PointerUpEvent
- PointerCancelEvent

Flutter también proporciona un pequeño conjunto de widgets para hacer gestos específicos y avanzados. Los widgets se enumeran a continuación:

- **Descartable** : admite gestos de desplazamiento para descartar el widget.
- **Arrastrable** : admite el gesto de arrastre para mover el widget.
- **LongPressDraggable** : admite el gesto de arrastre para mover un widget, cuando su widget principal también se puede arrastrar.
- **DragTarget** : acepta cualquier widget *arrastrable*
- **IgnorePointer** : oculta el widget y sus elementos **secundarios** del proceso de detección de gestos.
- **AbsorbPointer** : detiene el proceso de detección de gestos y, por lo tanto, cualquier widget superpuesto tampoco puede participar en el proceso de detección de gestos y, por lo tanto, no se genera ningún evento.
- **Desplazable** : **admite** el desplazamiento del contenido disponible dentro del widget.

Flutter - Gestión del Estado

La gestión del estado en una aplicación es uno de los procesos más importantes y necesarios en el ciclo de vida de una aplicación.

Consideremos una simple aplicación de carrito de compras.

- El usuario iniciará sesión con sus credenciales en la aplicación.
- Una vez que el usuario ha iniciado sesión, la aplicación debe conservar los detalles del usuario conectado en toda la pantalla.
- Nuevamente, cuando el usuario selecciona un producto y lo guarda en un carrito, la información del carrito debe persistir entre las páginas hasta que el usuario haya retirado el carrito.
- La información del usuario y su carrito en cualquier instancia se denomina estado de la aplicación en esa instancia.

Una gestión de estado se puede dividir en dos categorías en función de la duración del estado particular en una aplicación.

- Efímero : dura unos segundos, como el estado actual de una animación o una sola página, como la calificación actual de un producto. *Flutter* lo admite a través de `StatefulWidget`.
- estado de la aplicación : último para toda la aplicación, como detalles de usuario registrados, información del carrito, etc., *Flutter* lo admite a través de `scoped_model`.

Navegación y enrutamiento

En cualquier aplicación, navegar de una página / pantalla a otra define el flujo de trabajo de la aplicación. La forma en que se maneja la navegación de una aplicación se denomina Enrutamiento. Flutter proporciona una clase de enrutamiento básica: `MaterialPageRoute` y dos métodos: `Navigator.push` y `Navigator.pop`, para definir el flujo de trabajo de una aplicación.

MaterialPáginaRuta

MaterialPageRoute es un widget utilizado para representar su interfaz de usuario al reemplazar toda la pantalla con una animación específica de la plataforma.

```
MaterialPageRoute(builder: (context) => Widget())
```

Aquí, el constructor aceptará una función para construir su contenido al sustituir el contexto actual de la aplicación.

Navigation.push

Navigation.push se usa para navegar a una nueva pantalla usando el widget MaterialPageRoute.

```
Navigator.push( context, MaterialPageRoute(builder: (context)
=> Widget()), );
```

Navigation.pop

Navigation.pop se usa para navegar a la pantalla anterior.

```
Navigator.push(context);
```

Creemos una nueva aplicación para comprender mejor el concepto de navegación.

Cree una nueva aplicación Flutter en el estudio de Android, product_nav_app

- Copie la carpeta de activos de product_nav_app en product_state_app y agregue activos dentro del archivo pubspec.yaml.

```
flutter:
  assets:
    - assets/appimages/floppy.png
    - assets/appimages/iphone.png
    - assets/appimages/laptop.png
    - assets/appimages/pendrive.png
    - assets/appimages/pixel.png
    - assets/appimages/tablet.png
```

- Reemplace el código de inicio predeterminado (main.dart) con nuestro código de inicio.

```
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(
        primarySwatch: Colors.blue,
```

```

        ),
        home: MyHomePage(
          title: 'Product state demo home page'
        ),
      );
    }
  }
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: Text('Hello World',)
      ),
    );
  }
}

```

- Permítanos crear una clase de Producto para organizar la información del producto.

```

class Product {
  final String name;
  final String description;
  final int price;
  final String image;
  Product(this.name, this.description, this.price,
    this.image);
}

```

- Escribamos un método getProducts en la clase Product para generar nuestros registros ficticios de productos.

```

static List<Product> getProducts() {
  List<Product> items = <Product>[];

  items.add(
    Product(
      "Pixel",
      "Pixel is the most feature-full phone ever", 800,
      "pixel.png"
    )
  );
  items.add(
    Product(
      "Laptop",
      "Laptop is most productive development tool",
      2000, "
      laptop.png"
    )
  );
}

```

```

    )
  );
  items.add(
    Product(
      "Tablet",
      "Tablet is the most useful device ever for meeting",
      1500,
      "tablet.png"
    )
  );
  items.add(
    Product(
      "Pendrive",
      "Pendrive is useful storage medium",
      100,
      "pendrive.png"
    )
  );
  items.add(
    Product(
      "Floppy Drive",
      "Floppy drive is useful rescue storage medium",
      20,
      "floppy.png"
    )
  );
  return items;
}
import product.dart in main.dart
import 'Product.dart';

```

- Incluyamos nuestro nuevo widget, RatingBox.

```

class RatingBox extends StatefulWidget {
  @override
  _RatingBoxState createState() => _RatingBoxState();
}
class _RatingBoxState extends State<RatingBox> {
  int _rating = 0;
  void _setRatingAsOne() {
    setState(() {
      _rating = 1;
    });
  }
  void _setRatingAsTwo() {
    setState(() {
      _rating = 2;
    });
  }
  void _setRatingAsThree() {
    setState(() {
      _rating = 3;
    });
  }
}

```



```

Widget build(BuildContext context) {
  double _size = 20;
  print(_rating);
  return Row(
    mainAxisAlignment: MainAxisAlignment.end,
    crossAxisAlignment: CrossAxisAlignment.end,
    mainAxisAlignment: MainAxisAlignment.max,
    children: <Widget>[
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 1?
            Icon(
              Icons.star,
              size: _size,
            )
            : Icon(
              Icons.star_border,
              size: _size,
            )
          ),
          color: Colors.red[500],
          onPressed: _setRatingAsOne,
          iconSize: _size,
        ),
      ),
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 2?
            Icon(
              Icons.star,
              size: _size,
            )
            : Icon(
              Icons.star_border,
              size: _size,
            )
          ),
          color: Colors.red[500],
          onPressed: _setRatingAsTwo,
          iconSize: _size,
        ),
      ),
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 3 ?
            Icon(
              Icons.star,

```

```

        size: _size,
      ),
      : Icon(
        Icons.star_border,
        size: _size,
      ),
    ),
    color: Colors.red[500],
    onPressed: _setRatingAsThree,
    iconSize: _size,
  ),
],
);
}
}

```

- Modifiquemos nuestro widget ProductBox para que funcione con nuestra nueva clase de Producto.

```

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.item}) : super(key: key);
  final Product item;

  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2),
      height: 140,
      child: Card(
        child: Row(
          mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Image.asset("assets/appimages/" +
this.item.image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(this.item.name,
                      style: TextStyle(fontWeight:
FontWeight.bold)),
                    Text(this.item.description),
                    Text("Price: " +
this.item.price.toString()),
                    RatingBox(),
                  ],
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}

```

```

    ),
  ),
);
}
}

```

Permítanos reescribir nuestro widget `MyHomePage` para trabajar con el modelo de `Producto` y para enumerar todos los productos usando `ListView`.

```

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  final items = Product.getProducts();

  @override
  Widget build(BuildContext context) {
    return Scaffold( appBar: AppBar(title: Text("Product
Navigation")),
    body: ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        return GestureDetector(
          child: ProductBox(item: items[index]),
          onTap: () {
            Navigator.push(
              context, MaterialPageRoute(
                builder: (context) =>
ProductPage(item: items[index]),
              ),
            );
          },
        );
      },
    );
  },
);
}
}

```

Aquí, hemos utilizado `MaterialPageRoute` para navegar a la página de detalles del producto.

- Ahora, agreguemos `ProductPage` para mostrar los detalles del producto.

```

class ProductPage extends StatelessWidget {
  ProductPage({Key key, this.item}) : super(key: key);
  final Product item;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.item.name),
      ),
      body: Center(
        child: Container(

```

```

        padding: EdgeInsets.all(0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.start,
          crossAxisAlignment:
CrossAxisAlignment.start,
          children: <Widget>[
            Image.asset("assets/appimages/" +
this.item.image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(
                      this.item.name, style:
TextStyle(
                        fontWeight:
FontWeight.bold
                      )
                    ),
                    Text(this.item.description),
                    Text("Price: " +
this.item.price.toString()),
                    RatingBox(),
                  ],
                ),
              ),
            ),
          ],
        ),
      ),
    ),
  ],
);
}
}

```

El código completo de la aplicación es el siguiente:

```

import 'package:flutter/material.dart';
void main() => runApp(MyApp());

class Product {
  final String name;
  final String description;
  final int price;
  final String image;
  Product(this.name, this.description, this.price,
this.image);

  static List<Product> getProducts() {
    List<Product> items = <Product>[];
    items.add(

```

```

        Product(
            "Pixel",
            "Pixel is the most featureful phone ever",
            800,
            "pixel.png"
        )
    );
    items.add(
        Product(
            "Laptop",
            "Laptop is most productive development tool",
            2000,
            "laptop.png"
        )
    );
    items.add(
        Product(
            "Tablet",
            "Tablet is the most useful device ever for
meeting",
            1500,
            "tablet.png"
        )
    );
    items.add(
        Product(
            "Pendrive",
            "iPhone is the stylist phone ever",
            100,
            "pendrive.png"
        )
    );
    items.add(
        Product(
            "Floppy Drive",
            "iPhone is the stylist phone ever",
            20,
            "floppy.png"
        )
    );
    items.add(
        Product(
            "iPhone",
            "iPhone is the stylist phone ever",
            1000,
            "iphone.png"
        )
    );
    return items;
}
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.

```

```

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: MyHomePage(title: 'Product Navigation demo
home page'),
  );
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  final items = Product.getProducts();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Product Navigation")),
      body: ListView.builder(
        itemCount: items.length,
        itemBuilder: (context, index) {
          return GestureDetector(
            child: ProductBox(item: items[index]),
            onTap: () {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) =>
ProductPage(item: items[index]),
                ),
              );
            },
          );
        },
      );
    );
  }
}

class ProductPage extends StatelessWidget {
  ProductPage({Key key, this.item}) : super(key: key);
  final Product item;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.item.name),
      ),
      body: Center(
        child: Container(

```

```

        padding: EdgeInsets.all(0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.start,
          crossAxisAlignment:
CrossAxisAlignment.start,
          children: <Widget>[
            Image.asset("assets/appimages/" +
this.item.image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(this.item.name, style:
TextStyle(fontWeight: FontWeight.bold)),
                    Text(this.item.description),
                    Text("Price: " +
this.item.price.toString()),
                    RatingBox(),
                  ],
                ),
              ),
            ),
          ],
        ),
      ),
    ),
  ),
);
}
}

class RatingBox extends StatefulWidget {
  @override
  _RatingBoxState createState() => _RatingBoxState();
}

class _RatingBoxState extends State<RatingBox> {
  int _rating = 0;
  void _setRatingAsOne() {
    setState(() {
      _rating = 1;
    });
  }
  void _setRatingAsTwo() {
    setState(() {
      _rating = 2;
    });
  }
  void _setRatingAsThree() {
    setState(() {
      _rating = 3;
    });
  }
}

```

```

Widget build(BuildContext context) {
  double _size = 20;
  print(_rating);
  return Row(
    mainAxisAlignment: MainAxisAlignment.end,
    crossAxisAlignment: CrossAxisAlignment.end,
    mainAxisAlignment: MainAxisAlignment.max,
    children: <Widget>[
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 1 ? Icon(
              Icons.star,
              size: _size,
            )
            : Icon(
              Icons.star_border,
              size: _size,
            )
          ),
          color: Colors.red[500],
          onPressed: _setRatingAsOne,
          iconSize: _size,
        ),
      ),
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 2 ?
            Icon(
              Icons.star,
              size: _size,
            )
            : Icon(
              Icons.star_border,
              size: _size,
            )
          ),
          color: Colors.red[500],
          onPressed: _setRatingAsTwo,
          iconSize: _size,
        ),
      ),
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (
            _rating >= 3 ?
            Icon(
              Icons.star,
              size: _size,

```



```

        )
        : Icon(
            Icons.star_border,
            size: _size,
        )
    ),
    color: Colors.red[500],
    onPressed: _setRatingAsThree,
    iconSize: _size,
),
),
],
);
}
}

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.item}) : super(key: key);
  final Product item;

  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2),
      height: 140,
      child: Card(
        child: Row(
          mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Image.asset("assets/appimages/" +
this.item.image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(this.item.name, style:
TextStyle(fontWeight: FontWeight.bold)),
                    Text(this.item.description),
                    Text("Price: " +
this.item.price.toString()),
                    RatingBox(),
                  ],
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}

```

Ejecute la aplicación y haga clic en cualquiera de los elementos del producto. Mostrará la página de detalles relevantes. Podemos movernos a la página de inicio haciendo clic en el botón Atrás. La página de la lista de productos y la página de detalles del producto de la aplicación se muestran a continuación:





Flutter - Animación

La animación es un procedimiento complejo en cualquier aplicación móvil. A pesar de su complejidad, la Animación mejora la experiencia del usuario a un nuevo nivel y proporciona una rica interacción del usuario. Debido a su riqueza, la animación se convierte en una parte integral de la aplicación móvil moderna. Flutter Framework reconoce la importancia de la animación y proporciona un marco simple e intuitivo para desarrollar todo tipo de animaciones.

Introducción

La animación es un proceso de mostrar una serie de imágenes / imágenes en un orden particular dentro de una duración específica para dar una ilusión de movimiento. Los aspectos más importantes de la animación son los siguientes:

- La animación tiene dos valores distintos: valor inicial y valor final. La animación comienza desde el valor de *Inicio* y pasa por una serie de valores intermedios y finalmente termina en los valores de *Fin*. Por ejemplo, para animar un widget para que se desvanezca, el valor inicial será la opacidad total y el valor final será la opacidad cero.
- Los valores intermedios pueden ser de naturaleza lineal o no lineal (curva) y pueden configurarse. Comprenda que la animación funciona tal como está configurada. Cada configuración proporciona una sensación diferente a la animación. Por ejemplo, el desvanecimiento de un widget será de naturaleza lineal, mientras que el rebote de una pelota será de naturaleza no lineal.
- La duración del proceso de animación afecta la velocidad (lentitud o solidez) de la animación.
- La capacidad de controlar el proceso de animación, como comenzar la animación, detener la animación, repetir la animación para establecer el número de veces, invertir el proceso de animación, etc.
- En Flutter, el sistema de animación no hace ninguna animación real. En cambio, proporciona solo los valores requeridos en cada cuadro para representar las imágenes.

Clases basadas en animación

El sistema de animación Flutter se basa en objetos de animación. Las clases principales de animación y su uso son las siguientes:

Animación

Genera valores interpolados entre dos números durante una cierta duración. Las clases de animación más comunes son:

- **Animación <double>** : interpola valores entre dos números decimales
- **Animación <Color>** : interpola colores entre dos colores
- **Animación <Tamaño>** : interpolar tamaños entre dos tamaños

- **AnimationController** : objeto de animación especial para controlar la animación en sí. Genera nuevos valores cada vez que la aplicación está lista para un nuevo marco. Es compatible con la animación basada en lineal y el valor comienza desde 0.0 a 1.0

```
controller = AnimationController(duration: const
Duration(seconds: 2), vsync: this);
```

Aquí, el controlador controla la animación y la opción de duración controla la duración del proceso de animación. vsync es una opción especial utilizada para optimizar el recurso utilizado en la animación.

Animación curvada

Similar a AnimationController pero admite animación no lineal. CurvedAnimation se puede usar junto con el objeto Animation de la siguiente manera:

```
controller = AnimationController(duration: const
Duration(seconds: 2), vsync: this);
animation = CurvedAnimation(parent: controller, curve:
Curves.easeIn)
```

Interpolación <T>

Derivado de Animatable <T> y utilizado para generar números entre dos números distintos de 0 y 1. Se puede usar junto con el objeto Animación mediante el método animate y pasando el objeto Animación real.

```
AnimationController controller = AnimationController(
  duration: const Duration(milliseconds: 1000),
  vsync: this); Animation<int> customTween = IntTween(
  begin: 0, end: 255).animate(controller);
```

- Tween también se puede usar junto con CurvedAnimation como se muestra a continuación:

```
AnimationController controller = AnimationController(
  duration: const Duration(milliseconds: 500), vsync: this);
final Animation curve = CurvedAnimation(parent: controller,
curve: Curves.easeOut);
Animation<int> customTween = IntTween(begin: 0, end:
255).animate(curve);
```

Aquí, el controlador es el controlador de animación real. La curva proporciona el tipo de no linealidad y el CustomTween proporciona un rango personalizado de 0 a 255.

Flujo de trabajo de la animación Flutter

El flujo de trabajo de la animación es el siguiente:

- Defina e inicie el controlador de animación en initState del StatefulWidget.

```

AnimationController(duration: const Duration(seconds: 2),
vsync: this);
animation = Tween<double>(begin: 0, end:
300).animate(controller);
controller.forward();

```

- Agregue un oyente basado en animación, `addListener` para cambiar el estado del widget.

```

animation = Tween<double>(begin: 0, end:
300).animate(controller) ..addListener(() {
  setState(() {
    // The state that has changed here is the animation
    object's value.
  });
});

```

- Los widgets integrados, `AnimatedWidget` y `AnimatedBuilder` se pueden usar para omitir este proceso. Ambos widgets aceptan objetos de animación y obtienen los valores actuales necesarios para la animación.
- Obtenga los valores de animación durante el proceso de construcción del widget y luego aplíquelo para ancho, alto o cualquier propiedad relevante en lugar del valor original.

```

child: Container(
  height: animation.value,
  width: animation.value,
  child: <Widget>,
)

```

Aplicación de trabajo

Escribamos una aplicación simple basada en animación para comprender el concepto de animación en el marco Flutter.

- Cree una nueva aplicación *Flutter* en el estudio de Android, `product_animation_app`.
- Copie la carpeta de activos de `product_nav_app` a `product_animation_app` y agregue activos dentro del archivo `pubspec.yaml`.

```

flutter:
  assets:
    - assets/appimages/floppy.png
    - assets/appimages/iphone.png
    - assets/appimages/laptop.png
    - assets/appimages/pendrive.png
    - assets/appimages/pixel.png
    - assets/appimages/tablet.png

```

- Elimine el código de inicio predeterminado (`main.dart`).
- Añadir importación y función principal básica.

```

import 'package:flutter/material.dart';

```

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

- Cree el widget MyApp derivado de StatefulWidget.

```
class MyApp extends StatefulWidget {  
  _MyAppState createState() => _MyAppState();  
}
```

- Cree el widget _MyAppState e implemente initState y elimine, además del método de compilación predeterminado.

```
class _MyAppState extends State<MyApp> with  
SingleTickerProviderStateMixin {  
  Animation<double> animation;  
  AnimationController controller;  
  @override void initState() {  
    super.initState();  
    controller = AnimationController(  
      duration: const Duration(seconds: 10), vsync: this  
    );  
    animation = Tween<double>(begin: 0.0, end:  
1.0).animate(controller);  
    controller.forward();  
  }  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    controller.forward();  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(primarySwatch: Colors.blue,),  
      home: MyHomePage(title: 'Product layout demo home  
page', animation: animation),  
    );  
  }  
  @override  
  void dispose() {  
    controller.dispose();  
    super.dispose();  
  }  
}
```

Aquí,

- En el método initState, hemos creado un objeto controlador de animación (controlador), un objeto de animación (animación) y comenzamos la animación usando controller.forward.
- En el método de disposición, hemos dispuesto el objeto controlador de animación (controlador).
- En el método de compilación, envíe animación al widget MyHomePage a través del constructor. Ahora, el widget MyHomePage puede usar el objeto de animación para animar su contenido.
- Ahora, agregue el widget ProductBox

```

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.name, this.description,
this.price, this.image})
    : super(key: key);
  final String name;
  final String description;
  final int price;
  final String image;

  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2),
      height: 140,
      child: Card(
        child: Row(
          mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Image.asset("assets/appimages/" + image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(this.name, style:
                      TextStyle(fontWeight:
FontWeight.bold)),
                    Text(this.description),
                    Text("Price: " +
this.price.toString()),
                  ],
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}

```

- Cree un nuevo widget, MyAnimatedWidget para hacer una animación de desvanecimiento simple usando opacidad.

```

class MyAnimatedWidget extends StatelessWidget {
  MyAnimatedWidget({this.child, this.animation});

  final Widget child;
  final Animation<double> animation;

  Widget build(BuildContext context) => Center(
    child: AnimatedBuilder(

```



```

        animation: animation,
        builder: (context, child) => Container(
            child: Opacity(opacity: animation.value, child:
child),
        ),
        child: child),
    );
}

```

- Aquí, hemos utilizado AnimatedBuilder para hacer nuestra animación. AnimatedBuilder es un widget que construye su contenido mientras realiza la animación al mismo tiempo. Acepta un objeto de animación para obtener el valor de animación actual. Hemos utilizado el valor de animación, animation.value para establecer la opacidad del widget secundario. En efecto, el widget animará el widget hijo usando el concepto de opacidad.
- Finalmente, cree el widget MyHomePage y use el objeto de animación para animar cualquiera de sus contenidos.

```

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title, this.animation}) :
super(key: key);

  final String title;
  final Animation<double>
animation;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Product Listing")),body:
ListView(
  shrinkWrap: true,
  padding: const EdgeInsets.fromLTRB(2.0, 10.0,
2.0, 10.0),
  children: <Widget>[
    FadeTransition(
      child: ProductBox(
        name: "iPhone",
        description: "iPhone is the stylist
phone ever",
        price: 1000,
        image: "iphone.png"
      ), opacity: animation
    ),
    MyAnimatedWidget(child: ProductBox(
      name: "Pixel",
      description: "Pixel is the most featureful
phone ever",
      price: 800,
      image: "pixel.png"
    ), animation: animation),
    ProductBox(
      name: "Laptop",

```

```

        description: "Laptop is most productive
development tool",
        price: 2000,
        image: "laptop.png"
    ),
    ProductBox(
        name: "Tablet",
        description: "Tablet is the most useful
device ever for meeting",
        price: 1500,
        image: "tablet.png"
    ),
    ProductBox(
        name: "Pendrive",
        description: "Pendrive is useful storage
medium",
        price: 100,
        image: "pendrive.png"
    ),
    ProductBox(
        name: "Floppy Drive",
        description: "Floppy drive is useful rescue
storage medium",
        price: 20,
        image: "floppy.png"
    ),
  ],
),
);
}
}

```

Aquí, hemos utilizado FadeAnimation y MyAnimationWidget para animar los dos primeros elementos de la lista. FadeAnimation es una clase de animación incorporada, que utilizamos para animar a su hijo utilizando el concepto de opacidad.

- El código completo es el siguiente:

```

import 'package:flutter/material.dart';
void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> with
SingleTickerProviderStateMixin {
  Animation<double> animation;
  AnimationController controller;

  @override
  void initState() {
    super.initState();
    controller = AnimationController(

```

```

        duration: const Duration(seconds: 10), vsync: this);
        animation = Tween<double>(begin: 0.0, end:
1.0).animate(controller);
        controller.forward();
    }
    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        controller.forward();
        return MaterialApp(
            title: 'Flutter Demo', theme:
ThemeData(primarySwatch: Colors.blue,),
            home: MyHomePage(title: 'Product layout demo home
page', animation: animation,)
        );
    }
    @override
    void dispose() {
        controller.dispose();
        super.dispose();
    }
}

class MyHomePage extends StatelessWidget {
    MyHomePage({Key key, this.title, this.animation}):
super(key: key);
    final String title;
    final Animation<double> animation;

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text("Product Listing")),
            body: ListView(
                shrinkWrap: true,
                padding: const EdgeInsets.fromLTRB(2.0, 10.0,
2.0, 10.0),
                children: <Widget>[
                    FadeTransition(
                        child: ProductBox(
                            name: "iPhone",
                            description: "iPhone is the stylist
phone ever",
                            price: 1000,
                            image: "iphone.png"
                        ),
                        opacity: animation
                    ),
                    MyAnimatedWidget(
                        child: ProductBox(
                            name: "Pixel",
                            description: "Pixel is the most
featureful phone ever",
                            price: 800,

```

```

        image: "pixel.png"
      ),
      animation: animation
    ),
    ProductBox(
      name: "Laptop",
      description: "Laptop is most productive
development tool",
      price: 2000,
      image: "laptop.png"
    ),
    ProductBox(
      name: "Tablet",
      description: "Tablet is the most useful
device ever for meeting",
      price: 1500,
      image: "tablet.png"
    ),
    ProductBox(
      name: "Pendrive",
      description: "Pendrive is useful storage
medium",
      price: 100,
      image: "pendrive.png"
    ),
    ProductBox(
      name: "Floppy Drive",
      description: "Floppy drive is useful rescue
storage medium",
      price: 20,
      image: "floppy.png"
    ),
  ],
)
);
}
}

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.name, this.description,
this.price, this.image}) :
    super(key: key);
  final String name;
  final String description;
  final int price;
  final String image;
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2),
      height: 140,
      child: Card(
        child: Row(
          mainAxisAlignment:
MainAxisAlignment.spaceEvenly,

```

```

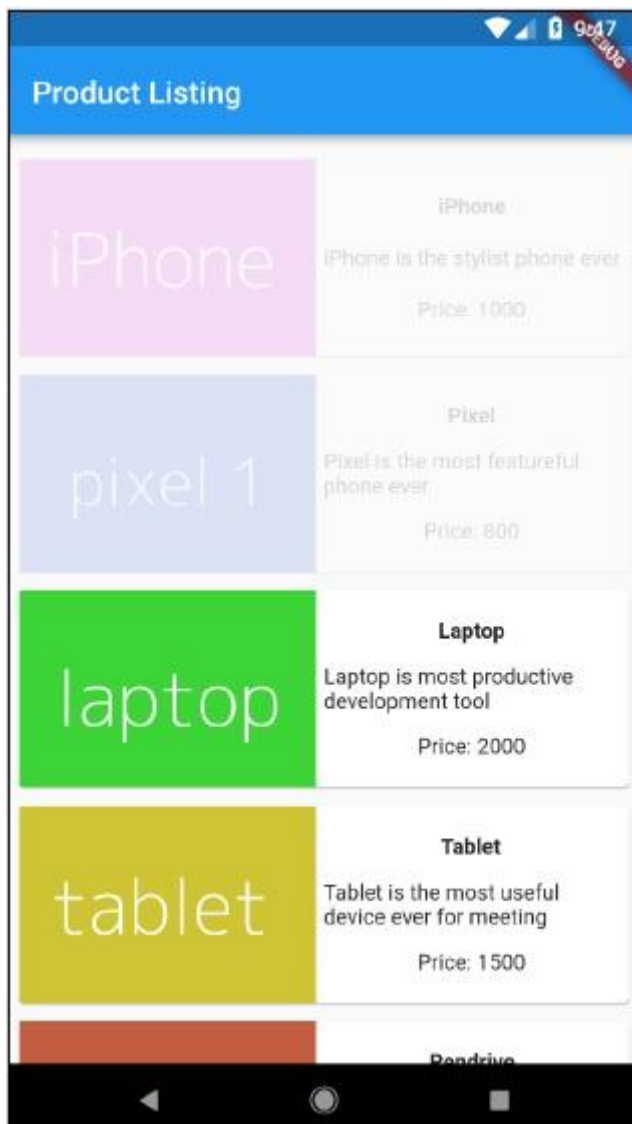
        children: <Widget>[
          Image.asset("assets/appimages/" + image),
          Expanded(
            child: Container(
              padding: EdgeInsets.all(5),
              child: Column(
                mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                children: <Widget>[
                  Text(
                    this.name, style: TextStyle(
                      fontWeight:
FontWeight.bold
                    ),
                  ),
                  Text(this.description), Text(
                    "Price: " +
this.price.toString()
                  ),
                ],
              ),
            ),
          ),
        ],
      ),
    ),
  );
}

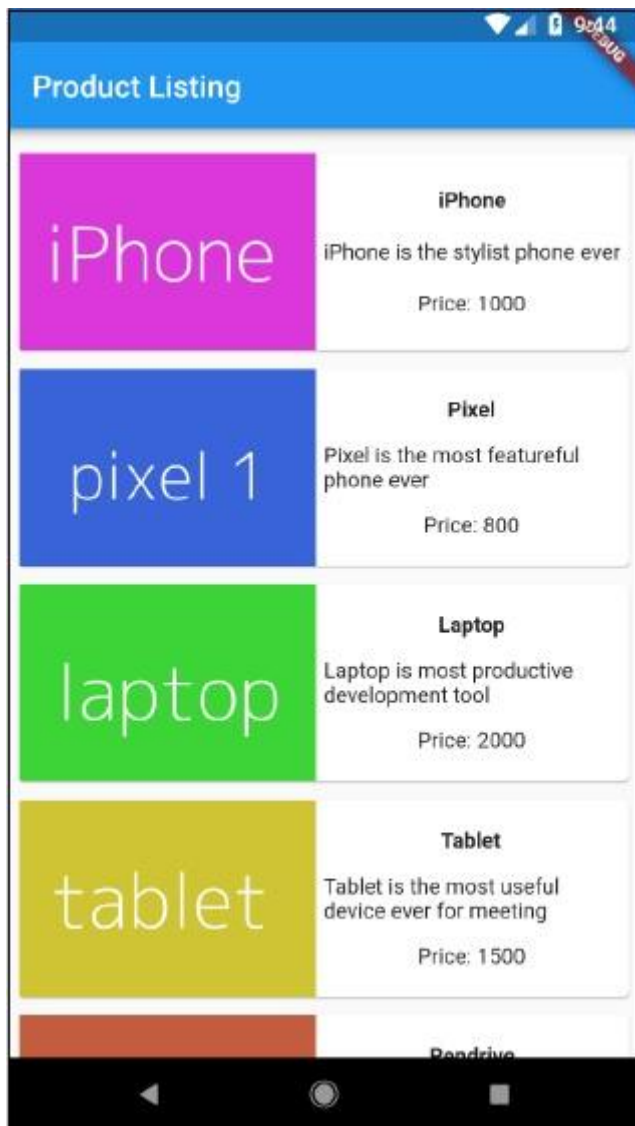
class MyAnimatedWidget extends StatelessWidget {
  MyAnimatedWidget({this.child, this.animation});
  final Widget child;
  final Animation<double> animation;

  Widget build(BuildContext context) => Center(
    child: AnimatedBuilder(
      animation: animation,
      builder: (context, child) => Container(
        child: Opacity(opacity: animation.value, child:
child),
      ),
      child: child
    ),
  );
}

```

- Compile y ejecute la aplicación para ver los resultados. La versión inicial y final de la aplicación es la siguiente:



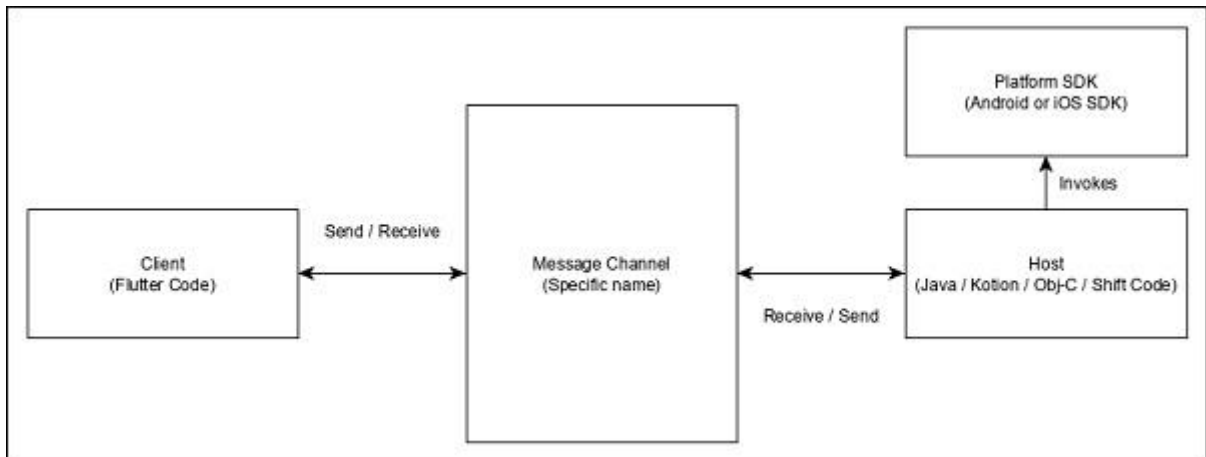


Flutter - Escribir código específico de Android

Flutter proporciona un marco general para acceder a la función específica de la plataforma. Esto permite al desarrollador ampliar la funcionalidad del marco *Flutter* utilizando código específico de la plataforma. Se puede acceder fácilmente a la funcionalidad específica de la plataforma como cámara, nivel de batería, navegador, etc. a través del marco.

La idea general de acceder al código específico de la plataforma es a través de un simple protocolo de mensajería. El código de Flutter, el cliente y el código de plataforma y el host se enlazan a un canal de mensajes común. El cliente envía un mensaje al host a través del canal de mensajes. El host escucha en el canal de mensajes, recibe el mensaje y realiza las funciones necesarias y, finalmente, devuelve el resultado al cliente a través del canal de mensajes.

La arquitectura del código específico de la plataforma se muestra en el diagrama de bloques que se muestra a continuación:



El protocolo de mensajería utiliza un códec de mensaje estándar (clase `StandardMessageCodec`) que admite la serialización binaria de valores similares a JSON, como números, cadenas, booleanos, etc. La serialización y la deserialización funcionan de manera transparente entre el cliente y el host.

Permítanos escribir una aplicación simple para abrir un navegador usando *Android SDK* y comprender cómo

- Cree una nueva aplicación Flutter en el estudio de Android, *flutter_browser_app*
- Reemplace el código `main.dart` con el siguiente código:

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open Browser'),
          onPressed: null,
        ),
      ),
    );
  }
}
```



```

    ),
  );
}
}

```

- Aquí, hemos creado un nuevo botón para abrir el navegador y establecer su método onPressed como nulo.
- Ahora, importe los siguientes paquetes:

```

import 'dart:async';
import 'package:flutter/services.dart';

```

- Aquí, services.dart incluye la funcionalidad para invocar código específico de la plataforma.
- Cree un nuevo canal de mensajes en el widget MyHomePage.

```

static const platform = const
MethodChannel('flutterapp.postparaprogramadores.com/browser')
;

```

- Escriba un método, _openBrowser para invocar el método específico de la plataforma, método openBrowser a través del canal de mensajes.

```

Future<void> _openBrowser() async {
  try {
    final int result = await platform.invokeMethod(
      'openBrowser', <String, String>{
        'url': "https://flutter.dev"
      }
    );
  }
  on PlatformException catch (e) {
    // Unable to open the browser
    print(e);
  }
}

```

Aquí, hemos utilizado platform.invokeMethod para invocar openBrowser (explicado en los próximos pasos). openBrowser tiene un argumento, url para abrir una url específica.

- Cambie el valor de la propiedad onPressed del RaisedButton de nulo a _openBrowser.

```

onPressed: _openBrowser,

```

- Abra MainActivity.java (dentro de la carpeta de Android) e importe la biblioteca requerida:

```

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;

```

```
import
io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugin.common.MethodChannel.Result;
import io.flutter.plugins.GeneratedPluginRegistrant;
```

- Escriba un método, openBrowser para abrir un navegador

```
private void openBrowser(MethodCall call, Result result,
String url) {
    Activity activity = this;
    if (activity == null) {
        result.error("ACTIVITY_NOT_AVAILABLE",
            "Browser cannot be opened without foreground
            activity", null);
        return;
    }
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse(url));

    activity.startActivity(intent);
    result.success((Object) true);
}
```

- Ahora, configure el nombre del canal en la clase MainActivity -

```
private static final String CHANNEL =
"flutterapp.postparaprogramadores.com/browser";
```

- Escriba código específico de Android para configurar el manejo de mensajes en el método onCreate -

```
new MethodChannel(getFlutterView(),
CHANNEL).setMethodCallHandler(
    new MethodCallHandler() {
        @Override
        public void onMethodCall(MethodCall call, Result result) {
            String url = call.argument("url");
            if (call.method.equals("openBrowser")) {
                openBrowser(call, result, url);
            } else {
                result.notImplemented();
            }
        }
    }
);
```

Aquí, hemos creado un canal de mensajes usando la clase MethodChannel y la clase MethodCallHandler para manejar el mensaje. onMethodCall es el método real responsable de llamar al código específico de la plataforma correcta al verificar el mensaje. El método onMethodCall extrae la url del mensaje y luego invoca el openBrowser solo cuando la llamada al método es openBrowser. De lo contrario, devuelve el método no implementado.

El código fuente completo de la aplicación es el siguiente:

main.dart

MainActivity.java

```

package
com.postparaprogramadores.flutterapp.flutter_browser_app;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel.Result;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
    private static final String CHANNEL =
"flutterapp.postparaprogramadores.com/browser";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        GeneratedPluginRegistrant.registerWith(this);
        new MethodChannel(getFlutterView(),
CHANNEL).setMethodCallHandler(
        new MethodCallHandler() {
            @Override
            public void onMethodCall(MethodCall call, Result
result) {
                String url = call.argument("url");
                if (call.method.equals("openBrowser")) {
                    openBrowser(call, result, url);
                } else {
                    result.notImplemented();
                }
            }
        }
    );
}

    private void openBrowser(MethodCall call, Result result,
String url) {
        Activity activity = this; if (activity == null) {
            result.error(
                "ACTIVITY_NOT_AVAILABLE", "Browser cannot be
opened without foreground activity", null
            );
            return;
        }
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse(url));
        activity.startActivity(intent);
        result.success((Object) true);
    }
}

```

main.dart

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

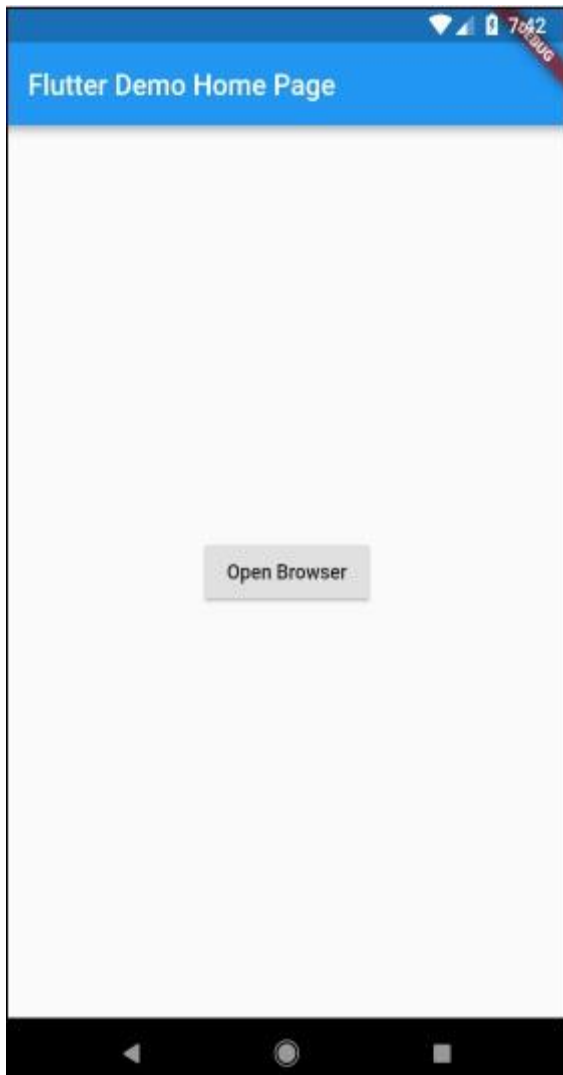
```

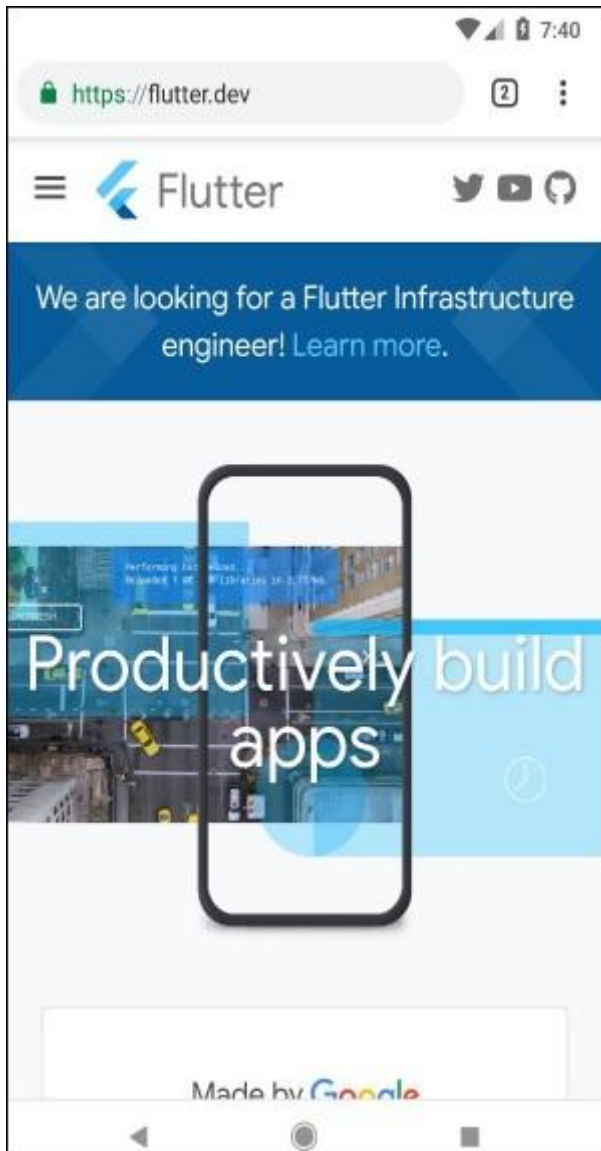
import 'dart:async';
import 'package:flutter/services.dart';

void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(
        title: 'Flutter Demo Home Page'
      ),
    );
  }
}
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  static const platform = const
MethodChannel('flutterapp.postparaprogramadores.com/browser')
;
  Future<void> _openBrowser() async {
    try {
      final int result = await
platform.invokeMethod('openBrowser', <String, String>{
        'url': "https://flutter.dev"
      });
    }
    on PlatformException catch (e) {
      // Unable to open the browser print(e);
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open Browser'),
          onPressed: _openBrowser,
        ),
      ),
    );
  }
}

```

Ejecute la aplicación y haga clic en el botón Abrir navegador y verá que se inicia el navegador. La aplicación del navegador: la página de inicio se muestra en la captura de pantalla aquí:





Flutter - Escribir código específico de iOS

El acceso al código específico de iOS es similar al de la plataforma Android, excepto que usa lenguajes específicos de iOS: Objective-C o Swift e iOS SDK. De lo contrario, el concepto es el mismo que el de la plataforma Android.

Escribamos también la misma aplicación que en el capítulo anterior para la plataforma iOS.

- Permítanos crear una nueva aplicación en Android Studio (macOS), *flutter_browser_ios_app*
- Siga los pasos 2 - 6 como en el capítulo anterior.
- Inicie XCode y haga clic en **Archivo** → **Abrir**
- Elija el proyecto xcode en el directorio ios de nuestro proyecto flutter.
- Abra AppDelegate.m en **Runner** → Ruta del **corredor** . Contiene el siguiente código:

```
#include "AppDelegate.h"
#include "GeneratedPluginRegistrant.h"
```

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
    // [GeneratedPluginRegistrant
registerWithRegistry:self];
    // Override point for customization after application
launch.
    return [super application:application
    didFinishLaunchingWithOptions:launchOptions];
}
@end
```

- Hemos agregado un método, openBrowser para abrir el navegador con la URL especificada. Acepta un solo argumento, url.

```
- (void)openBrowser:(NSString *)urlString {
    NSURL *url = [NSURL URLWithString:urlString];
    UIApplication *application = [UIApplication
sharedApplication];
    [application openURL:url];
}
```

- En el método didFinishLaunchingWithOptions, busque el controlador y configúrelo en la variable del controlador.

```
FlutterViewController* controller =
(FlutterViewController*)self.window.rootViewController;
```

- En el método didFinishLaunchingWithOptions, configure el canal del navegador como flutterapp.postparaprogramadores.com/browse -

```
FlutterMethodChannel* browserChannel = [
FlutterMethodChannel methodChannelWithName:
@"flutterapp.postparaprogramadores.com/browse"
binaryMessenger:controller];
```

- Cree una variable, weakSelf y establezca la clase actual:

```
__weak typeof(self) weakSelf = self;
```

- Ahora, implemente setMethodCallHandler. Llame a openBrowser haciendo coincidir call.method. Obtenga url invocando call.arguments y páselo mientras llama a openBrowser.

```
[browserChannel setMethodCallHandler:^(FlutterMethodCall*
call, FlutterResult result) {
    if ([@"openBrowser" isEqualToString:call.method]) {
        NSString *url = call.arguments[@"url"];
        [weakSelf openBrowser:url];
    } else { result(FlutterMethodNotImplemented); }
}];
```

- El código completo es el siguiente:

```
#include "AppDelegate.h"
#include "GeneratedPluginRegistrant.h"
```

```

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    // custom code starts
    FlutterViewController* controller =
(FlutterViewController*)self.window.rootViewController;
    FlutterMethodChannel* browserChannel = [
        FlutterMethodChannel methodChannelWithName:
            @"flutterapp.postparaprogramadores.com /browser"
binaryMessenger:controller];

    __weak typeof(self) weakSelf = self;
    [browserChannel setMethodCallHandler:^(
        FlutterMethodCall* call, FlutterResult result) {

        if ([@"openBrowser" isEqualToString:call.method]) {
            NSString *url = call.arguments[@"url"];
            [weakSelf openBrowser:url];
        } else { result(FlutterMethodNotImplemented); }
    }]];
    // custom code ends
    [GeneratedPluginRegistrant registerWithRegistry:self];

    // Override point for customization after application
    launch.
    return [super application:application
        didFinishLaunchingWithOptions:launchOptions];
}

- (void)openBrowser:(NSString *)urlString {
    NSURL *url = [NSURL URLWithString:urlString];
    UIApplication *application = [UIApplication
        sharedApplication];
    [application openURL:url];
}

@end

```

- Configuración de proyecto abierta.
- Vaya a **Capacidades** y habilite los **modos de fondo** .
- Agregue *** Recuperación de fondo y Notificación remota **** .
- Ahora, ejecuta la aplicación. Funciona de manera similar a la versión de Android, pero el navegador Safari se abrirá en lugar de Chrome.

Flutter - Introducción al paquete

La forma en que Dart organiza y comparte un conjunto de funcionalidades es a través de Package. Dart Package es simplemente bibliotecas o módulos compartibles. En general, el paquete Dart es el mismo que el de la aplicación Dart, excepto que el paquete Dart no tiene un punto de entrada de la aplicación, main.

La estructura general del paquete (considere un paquete de demostración, `my_demo_package`) es la siguiente:

- **lib / src / *** - Archivos privados de código Dart.
- **lib / my_demo_package.dart** : archivo de código de Dart principal. Se puede importar a una aplicación como:

```
import 'package:my_demo_package/my_demo_package.dart'
```

- Se puede exportar otro archivo de código privado al archivo de código principal (`my_demo_package.dart`), si es necesario, como se muestra a continuación:

```
export src/my_private_code.dart
```

- **lib / *** : cualquier número de archivos de código Dart dispuestos en cualquier estructura de carpetas personalizada. Se puede acceder al código como,

```
import
```

```
'package:my_demo_package/custom_folder/custom_file.dart'
```

- **pubspec.yaml** : especificación del proyecto, igual que la de la aplicación,

Todos los archivos de códigos Dart en el Paquete son simplemente clases Dart y no tiene ningún requisito especial para que un código Dart lo incluya en un Paquete.

Tipos de paquetes

Dado que los paquetes Dart son básicamente una pequeña colección de funcionalidades similares, se pueden clasificar en función de su funcionalidad.

Paquete de dardos

Código genérico de Dart, que se puede utilizar en entornos web y móviles. Por ejemplo, `english_words` es uno de esos paquetes que contiene alrededor de 5000 palabras y tiene funciones básicas de utilidad como sustantivos (enumerar sustantivos en inglés), sílabas (especificar el número de sílabas en una palabra).

Paquete Flutter

Código genérico de Dart, que depende del marco Flutter y solo se puede usar en un entorno móvil. Por ejemplo, `fluro` es un enrutador personalizado para flutter. Depende del marco Flutter.

Flutter Plugin

Código genérico de Dart, que depende del marco Flutter, así como del código de la plataforma subyacente (SDK de Android o SDK de iOS). Por ejemplo, la cámara es un complemento para interactuar con la cámara del dispositivo. Depende del marco Flutter y del marco subyacente para acceder a la cámara.

Usando un paquete de dardos

Los paquetes Dart se alojan y publican en el servidor en vivo, <https://pub.dartlang.org>. Además, Flutter proporciona una herramienta simple, pub para administrar paquetes de Dart en la aplicación. Los pasos necesarios para usar como paquete son los siguientes:

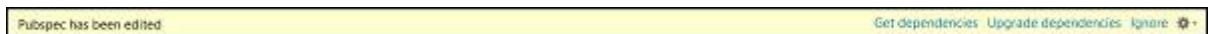
- Incluya el nombre del paquete y la versión necesaria en pubspec.yaml como se muestra a continuación:

```
dependencies: english_words: ^3.1.5
```

- El último número de versión se puede encontrar comprobando el servidor en línea.
- Instale el paquete en la aplicación utilizando el siguiente comando:

```
flutter packages get
```

- Mientras se desarrolla en el estudio de Android, Android Studio detecta cualquier cambio en pubspec.yaml y muestra una alerta de paquete de estudio de Android al desarrollador como se muestra a continuación:



- Los paquetes Dart se pueden instalar o actualizar en Android Studio usando las opciones del menú.
- Importe el archivo necesario con el comando que se muestra a continuación y comience a trabajar:

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

- Use cualquier método disponible en el paquete,

```
nouns.take(50).forEach(print);
```

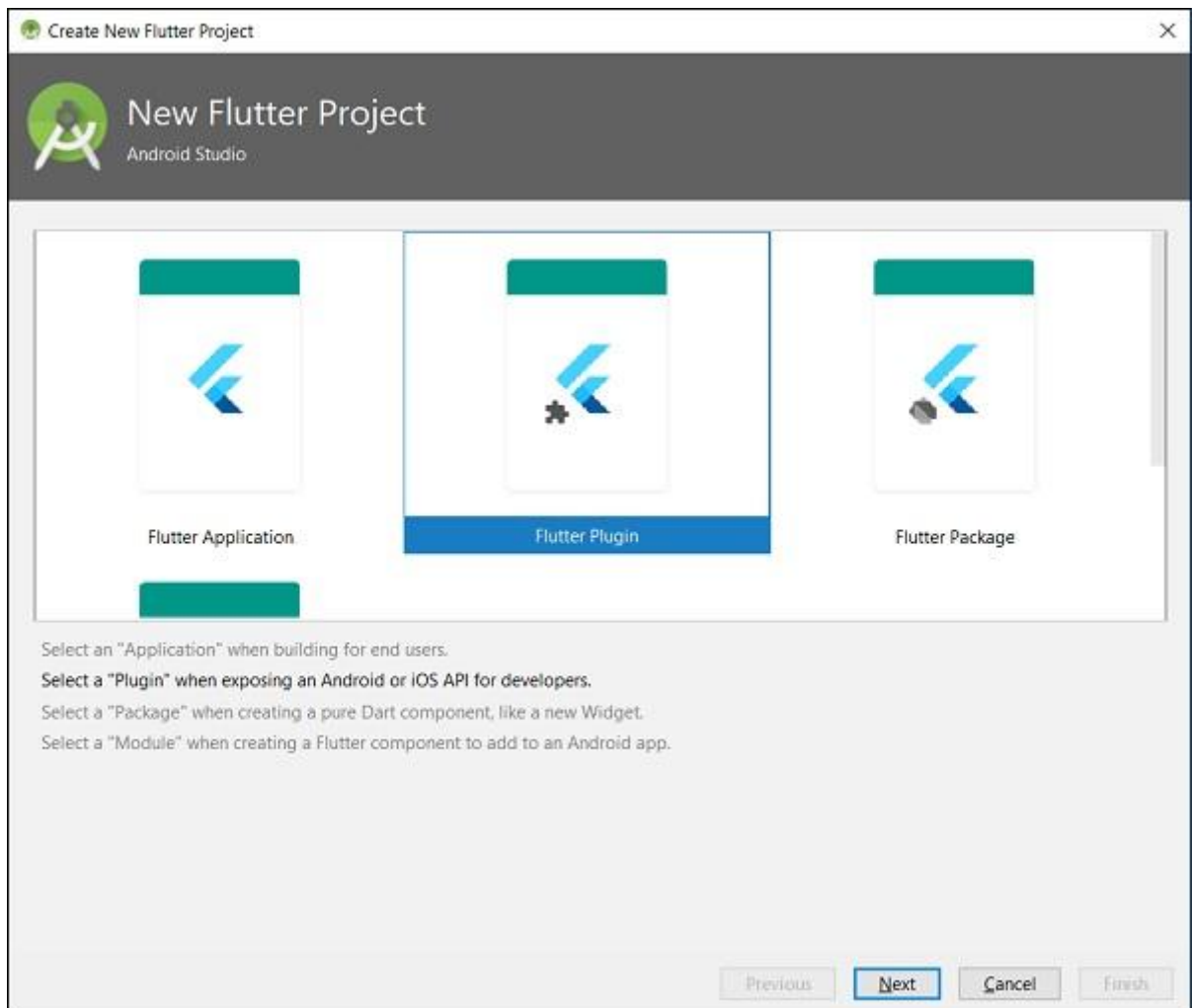
- Aquí, hemos utilizado la función de sustantivos para obtener e imprimir las 50 palabras principales.

Desarrollar un paquete de complementos Flutter

Desarrollar un complemento Flutter es similar a desarrollar una aplicación Dart o un paquete Dart. La única excepción es que el complemento utilizará la API del sistema (Android o iOS) para obtener la funcionalidad específica de la plataforma requerida.

Como ya hemos aprendido cómo acceder al código de la plataforma en los capítulos anteriores, desarrollemos un complemento simple, my_browser, para comprender el proceso de desarrollo del complemento. La funcionalidad del complemento my_browser es permitir que la aplicación abra el sitio web dado en el navegador específico de la plataforma.

- Inicia Android Studio.
- Haga clic en **Archivo** → **Nuevo proyecto Flutter** y seleccione la opción Complemento Flutter.
- Puede ver una ventana de selección del complemento Flutter como se muestra aquí:



- Ingrese my_browser como nombre del proyecto y haga clic en Siguiente.
- Ingrese el nombre del complemento y otros detalles en la ventana como se muestra aquí:

Create New Flutter Project

New Flutter Plugin
Android Studio

Configure the new Flutter plugin

Project name
my_browser

Flutter SDK path
E:\flutter ... [Install SDK...](#)

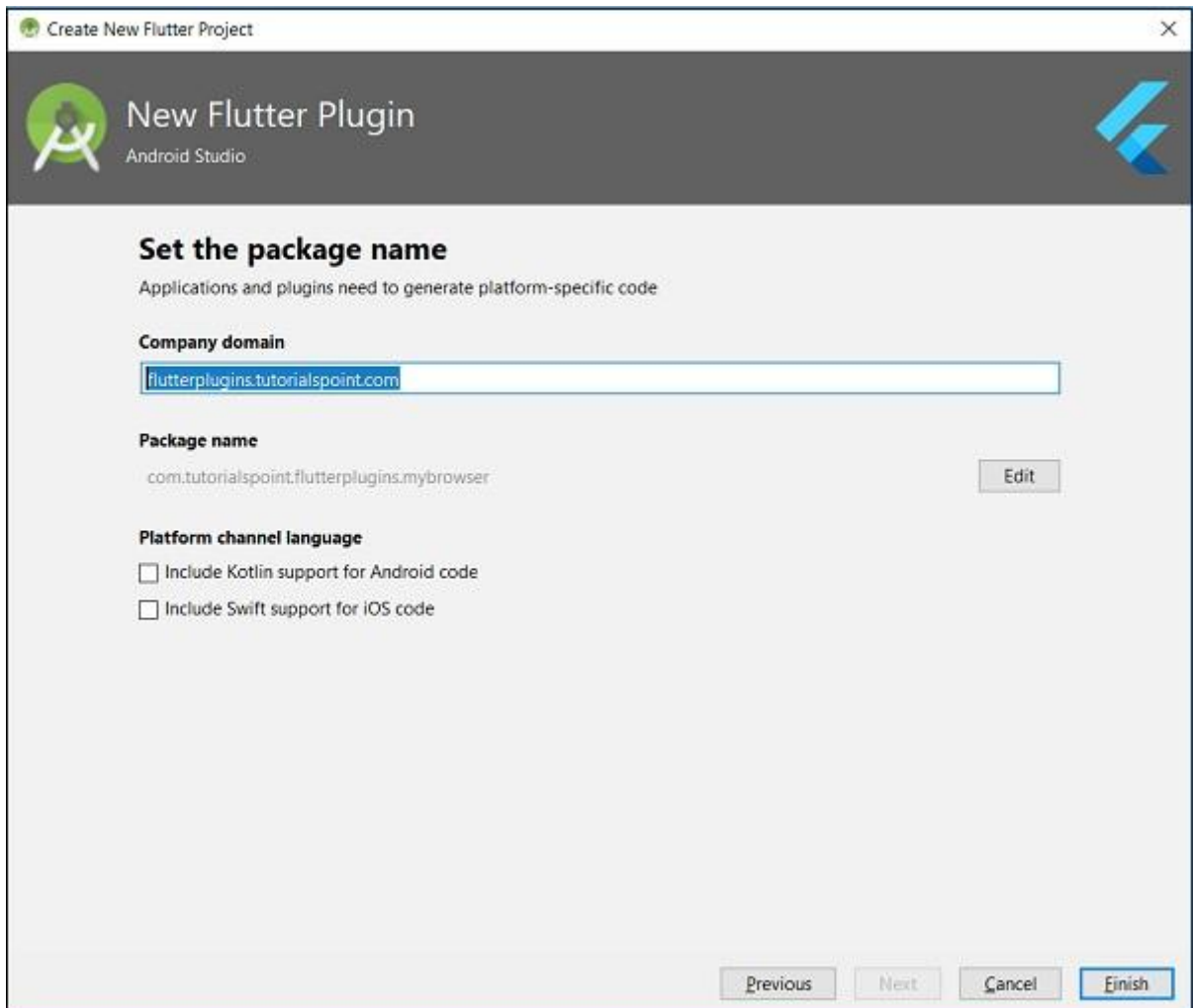
Project location
D:\dev\code\FlutterProjects ...

Description
A new Flutter plugin.

☐ Create project offline

Previous Next Cancel Finish

- Ingrese el dominio de la compañía, flutterplugins.postparaprogramadores.com en la ventana que se muestra a continuación y luego haga clic en **Finalizar** . Generará un código de inicio para desarrollar nuestro nuevo complemento.



- Abra el archivo my_browser.dart y escriba un método, openBrowser para invocar el método openBrowser específico de la plataforma.

```
Future<void> openBrowser(String urlString) async {  
  try {  
    final int result = await _channel.invokeMethod(  
      'openBrowser', <String, String>{ 'url': urlString }  
    );  
  }  
  on PlatformException catch (e) {  
    // Unable to open the browser print(e);  
  }  
}
```

- Abra el archivo MyBrowserPlugin.java e importe las siguientes clases:

```
import android.app.Activity;  
import android.content.Intent;  
import android.net.Uri;  
import android.os.Bundle;
```

- Aquí, tenemos que importar la biblioteca requerida para abrir un navegador desde Android.
- Agregue una nueva variable privada mRegistrar de tipo Registrar en la clase MyBrowserPlugin.

```
private final Registrar mRegistrar;
```

- Aquí, Registrar se usa para obtener información de contexto del código de invocación.
- Agregue un constructor para configurar Registrar en la clase MyBrowserPlugin.

```
private MyBrowserPlugin(Registrar registrar) {  
    this.mRegistrar = registrar;  
}
```

- Cambie registerWith para incluir nuestro nuevo constructor en la clase MyBrowserPlugin.

```
public static void registerWith(Registrar registrar) {  
    final MethodChannel channel = new  
MethodChannel(registrar.messenger(), "my_browser");  
    MyBrowserPlugin instance = new MyBrowserPlugin(registrar);  
    channel.setMethodCallHandler(instance);  
}
```

- Cambie onMethodCall para incluir el método openBrowser en la clase MyBrowserPlugin.

```
@Override  
public void onMethodCall(MethodCall call, Result result) {  
    String url = call.argument("url");  
    if (call.method.equals("getPlatformVersion")) {  
        result.success("Android " +  
android.os.Build.VERSION.RELEASE);  
    }  
    else if (call.method.equals("openBrowser")) {  
        openBrowser(call, result, url);  
    } else {  
        result.notImplemented();  
    }  
}
```

- Write the platform specific openBrowser method to access browser in MyBrowserPlugin class.

```
private void openBrowser(MethodCall call, Result result,  
String url) {  
    Activity activity = mRegistrar.activity();  
    if (activity == null) {  
        result.error("ACTIVITY_NOT_AVAILABLE",  
"Browser cannot be opened without foreground activity",  
null);  
        return;  
    }  
    Intent intent = new Intent(Intent.ACTION_VIEW);  
    intent.setData(Uri.parse(url));  
    activity.startActivity(intent);  
    result.success((Object) true);  
}
```

- The complete source code of the my_browser plugin is as follows –

my_browser.dart

```
import 'dart:async';
import 'package:flutter/services.dart';

class MyBrowser {
  static const MethodChannel _channel = const
MethodChannel('my_browser');
  static Future<String> get platformVersion async {
    final String version = await
_channel.invokeMethod('getPlatformVersion'); return version;
  }
  Future<void> openBrowser(String urlString) async {
    try {
      final int result = await _channel.invokeMethod(
        'openBrowser', <String, String>{'url':
urlString});
    }
    on PlatformException catch (e) {
      // Unable to open the browser print(e);
    }
  }
}
```

MyBrowserPlugin.java

```
package com.postparaprogramadores.flutterplugins.my_browser;

import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import
io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugin.common.MethodChannel.Result;
import io.flutter.plugin.common.PluginRegistry.Registrar;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

/** MyBrowserPlugin */
public class MyBrowserPlugin implements MethodCallHandler {
  private final Registrar mRegistrar;
  private MyBrowserPlugin(Registrar registrar) {
    this.mRegistrar = registrar;
  }
  /** Plugin registration. */
  public static void registerWith(Registrar registrar) {
    final MethodChannel channel = new MethodChannel(
      registrar.messenger(), "my_browser");
    MyBrowserPlugin instance = new
MyBrowserPlugin(registrar);
    channel.setMethodCallHandler(instance);
  }
  @Override
```

```

public void onMethodCall(MethodCall call, Result result) {
    String url = call.argument("url");
    if (call.method.equals("getPlatformVersion")) {
        result.success("Android " +
android.os.Build.VERSION.RELEASE);
    }
    else if (call.method.equals("openBrowser")) {
        openBrowser(call, result, url);
    } else {
        result.notImplemented();
    }
}

private void openBrowser(MethodCall call, Result result,
String url) {
    Activity activity = mRegistrar.activity();
    if (activity == null) {
        result.error("ACTIVITY_NOT_AVAILABLE",
            "Browser cannot be opened without foreground
activity", null);
        return;
    }
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse(url));
    activity.startActivity(intent);
    result.success((Object) true);
}
}

```

- Create a new project, *my_browser_plugin_test* to test our newly created plugin.
- Open pubspec.yaml and set my_browser as a plugin dependency.

```

dependencies:
  flutter:
    sdk: flutter
  my_browser:
    path: ../my_browser

```

- Android studio will alert that the pubspec.yaml is updated as shown in the Android studio package alert given below –

Pubspec has been edited Get dependencies Upgrade dependencies Ignore ⚙

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Open main.dart and include my_browser plugin as below –

```
import 'package:my_browser/my_browser.dart';
```

- Call the openBrowser function from my_browser plugin as shown below –

```

onPressed: () =>
MyBrowser().openBrowser("https://flutter.dev"),

```

- The complete code of the main.dart is as follows –

```

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

```



```

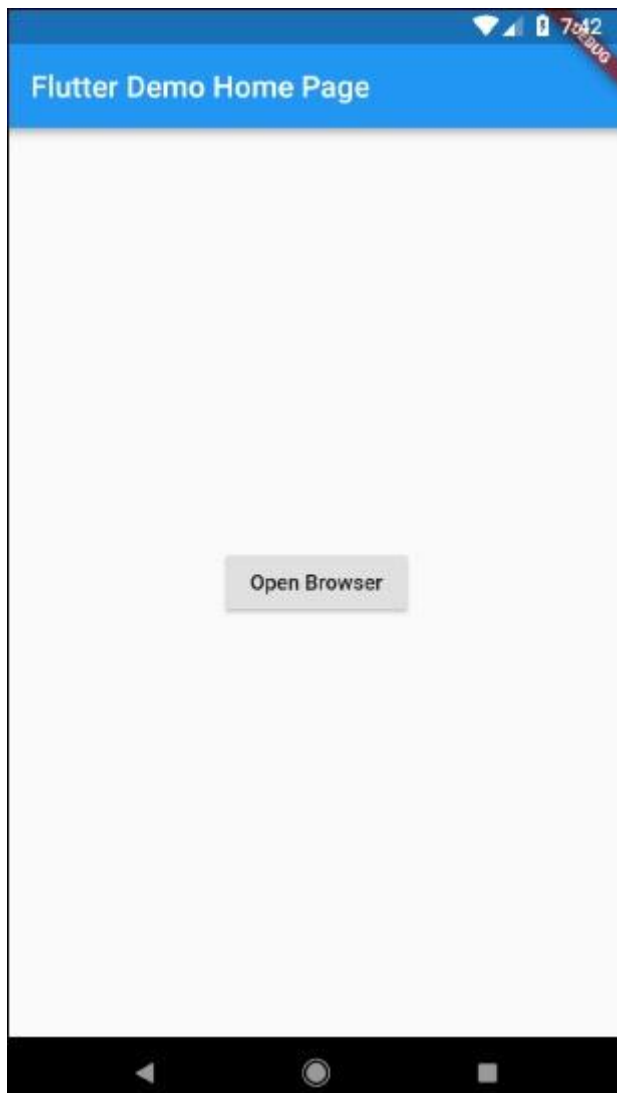
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(
        title: 'Flutter Demo Home Page'
      ),
    );
  }
}

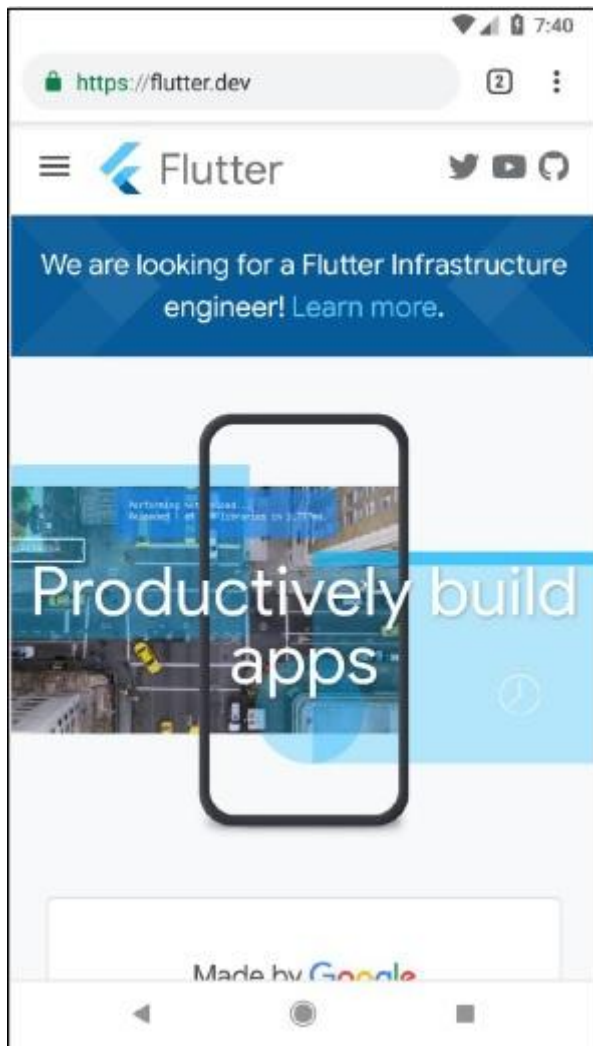
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open Browser'),
          onPressed: () =>
MyBrowser().openBrowser("https://flutter.dev"),
        ),
      ),
    );
  }
}

```

- Run the application and click the Open Browser button and see that the browser is launched. You can see a Browser app - Home page as shown in the screenshot shown below –



You can see a Browser app – Browser screen as shown in the screenshot shown below –



Flutter - Accessing REST API

Flutter provides http package to consume HTTP resources. http is a Future-based library and uses await and async features. It provides many high level methods and simplifies the development of REST based mobile applications.

Basic Concepts

http package provides a high level class and http to do web requests.

- http class provides functionality to perform all types of HTTP requests.
- http methods accept a url, and additional information through Dart Map (post data, additional headers, etc.,). It requests the server and collects the response back in async/await pattern. For example, the below code reads the data from the specified url and print it in the console.

```
print(await http.read('https://flutter.dev/'));
```

Some of the core methods are as follows –

- **read** – Request the specified url through GET method and return back the response as Future<String>
- **get** – Request the specified url through GET method and return back the response as Future<Response>. Response is a class holding the response information.
- **post** – Request the specified url through POST method by posting the supplied data and return back the response as Future<Response>
- **put** – Request the specified url through PUT method and return back the response as Future <Response>
- **head** – Request the specified url through HEAD method and return back the response as Future<Response>
- **delete** – Request the specified url through DELETE method and return back the response as Future<Response>

http also provides a more standard HTTP client class, client. client supports persistent connection. It will be useful when a lot of request to be made to a particular server. It needs to be closed properly using close method. Otherwise, it is similar to http class. The sample code is as follows –

```
var client = new http.Client();
try {
    print(await client.get('https://flutter.dev/'));
}
finally {
    client.close();
}
```

Accessing Product service API

Let us create a simple application to get product data from a web server and then show the products using *ListView*.

- Create a new *Flutter* application in Android studio, *product_rest_app*.
- Replace the default startup code (main.dart) with our *product_nav_app* code.

- Copy the assets folder from *product_nav_app* to *product_rest_app* and add assets inside the pubspec.yaml file.

```
flutter:
  assets:
    - assets/appimages/floppy.png
    - assets/appimages/iphone.png
    - assets/appimages/laptop.png
    - assets/appimages/pendrive.png
    - assets/appimages/pixel.png
    - assets/appimages/tablet.png
```

- Configure http package in the pubspec.yaml file as shown below –

```
dependencies:
  http: ^0.12.0+2
```

- Here, we will use the latest version of the http package. Android studio will send a package alert that the pubspec.yaml is updated.

Pubspec has been edited Get dependencies Upgrade dependencies Ignore

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Import http package in the main.dart file –

```
import 'dart:async';
import 'dart:convert';
import 'package:http/http.dart' as http;
```

- Create a new JSON file, products.json with product information as shown below –

```
[
  {
    "name": "iPhone",
    "description": "iPhone is the stylist phone ever",
    "price": 1000,
    "image": "iphone.png"
  },
  {
    "name": "Pixel",
    "description": "Pixel is the most feature phone ever",
    "price": 800,
    "image": "pixel.png"
  },
  {
    "name": "Laptop",
    "description": "Laptop is most productive development tool",
    "price": 2000,
    "image": "laptop.png"
  },
  {
    "name": "Tablet",
    "description": "Tablet is the most useful device ever for meeting",
    "price": 1500,
    "image": "tablet.png"
  },
  {
    "name": "Pendrive",
    "description": "Pendrive is useful storage medium",
    "price": 100,
    "image": "pendrive.png"
  }
]
```

```

    },
    {
      "name": "Floppy Drive",
      "description": "Floppy drive is useful rescue storage medium",
      "price": 20,
      "image": "floppy.png"
    }
  ]

```

- Create a new folder, JSONWebServer and place the JSON file, products.json.
- Run any web server with JSONWebServer as its root directory and get its web path. For example, `http://192.168.184.1:8000/products.json`. We can use any web server like apache, nginx etc.,
- The easiest way is to install node based http-server application. Follow the steps given below to install and run http- server application
 - Install Nodejs application (nodejs.org)
 - Go to JSONWebServer folder.

```
cd /path/to/JSONWebServer
```

- Install http-server package using npm.

```
npm install -g http-server
```

- Now, run the server.

```
http-server . -p 8000
```

```
Starting up http-server, serving .
```

```
Available on:
```

```
http://192.168.99.1:8000
```

```
http://127.0.0.1:8000
```

```
Hit CTRL-C to stop the server
```

- Create a new file, Product.dart in the lib folder and move the Product class into it.
- Write a factory constructor in the Product class, Product.fromMap to convert mapped data Map into the Product object. Normally, JSON file will be converted into Dart Map object and then, converted into relevant object (Product).

```

factory Product.fromJson(Map<String, dynamic> data) {
  return Product(
    data['name'],
    data['description'],
    data['price'],
    data['image'],
  );
}

```

- The complete code of the Product.dart is as follows –

```

class Product {
  final String name;
  final String description;
  final int price;
  final String image;

  Product(this.name, this.description, this.price, this.image);
  factory Product.fromMap(Map<String, dynamic> json) {
    return Product(
      json['name'],

```

```

        json['description'],
        json['price'],
        json['image'],
    );
}
}

```

- Write two methods – `parseProducts` and `fetchProducts` - in the main class to fetch and load the product information from web server into the `List<Product>` object.

```

List<Product> parseProducts(String responseBody) {
    final parsed = json.decode(responseBody).cast<Map<String,
dynamic>>();
    return parsed.map<Product>((json)
=>Product.fromJson(json)).toList();
}
Future<List<Product>> fetchProducts() async {
    final response = await
http.get('http://192.168.1.2:8000/products.json');
    if (response.statusCode == 200) {
        return parseProducts(response.body);
    } else {
        throw Exception('Unable to fetch products from the REST API');
    }
}

```

- Note the following points here –
 - Future is used to lazy load the product information. Lazy loading is a concept to defer the execution of the code until it is necessary.
 - `http.get` is used to fetch the data from the Internet.
 - `json.decode` is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into `List<Product>` using `fromMap` of the Product class.
 - In `MyApp` class, add new member variable, `products` of type `Future<Product>` and include it in constructor.

```

class MyApp extends StatelessWidget {
    final Future<List<Product>> products;
    MyApp({Key key, this.products}) : super(key: key);
    ...
}

```

- In `MyHomePage` class, add new member variable `products` of type `Future<Product>` and include it in constructor. Also, remove `items` variable and its relevant method, `getProducts` method call. Placing the `products` variable in constructor. It will allow to fetch the products from Internet only once when the application is first started.

```

class MyHomePage extends StatelessWidget {
    final String title;
    final Future<ListList<Product>> products;
    MyHomePage({Key key, this.title, this.products}) : super(key: key);
    ...
}

```

- Change the home option (`MyHomePage`) in the build method of `MyApp` widget to accommodate above changes –

```

home: MyHomePage(title: 'Product Navigation demo home page', products:
products),

```

- Change the main function to include `Future<Product>` arguments –

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

- Create a new widget, `ProductBoxList` to build the product list in the home page.

```
class ProductBoxList extends StatelessWidget {
  final List<Product> items;
  ProductBoxList({Key key, this.items});

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        return GestureDetector(
          child: ProductBox(item: items[index]),
          onTap: () {
            Navigator.push(
              context, MaterialPageRoute(
                builder: (context) => ProductPage(item:
items[index]),
              ),
            );
          },
        );
      },
    );
  }
}
```

Note that we used the same concept used in Navigation application to list the product except it is designed as a separate widget by passing products (object) of type `List<Product>`.

- Finally, modify the `MyHomePage` widget's build method to get the product information using Future option instead of normal method call.

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Product Navigation")),
    body: Center(
      child: FutureBuilder<List<Product>>(
        future: products, builder: (context, snapshot) {
          if (snapshot.hasError) print(snapshot.error);
          return snapshot.hasData ? ProductBoxList(items:
snapshot.data)

          // return the ListView widget :
          Center(child: CircularProgressIndicator());
        },
      ),
    ),
  );
}
```

- Here note that we used `FutureBuilder` widget to render the widget. `FutureBuilder` will try to fetch the data from it's future property (of type `Future<List<Product>>`). If future property returns data, it will render the widget using `ProductBoxList`, otherwise throws an error.
- The complete code of the `main.dart` is as follows –

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



```

import 'dart:convert';
import 'package:http/http.dart' as http;
import 'Product.dart';

void main() => runApp(MyApp(products: fetchProducts()));

List<Product> parseProducts(String responseBody) {
  final parsed = json.decode(responseBody).cast<Map<String,
dynamic>>();
  return parsed.map<Product>((json) =>
Product.fromMap(json)).toList();
}
Future<List<Product>> fetchProducts() async {
  final response = await
http.get('http://192.168.1.2:8000/products.json');
  if (response.statusCode == 200) {
    return parseProducts(response.body);
  } else {
    throw Exception('Unable to fetch products from the REST API');
  }
}
class MyApp extends StatelessWidget {
  final Future<List<Product>> products;
  MyApp({Key key, this.products}) : super(key: key);

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Product Navigation demo home page',
products: products),
    );
  }
}
class MyHomePage extends StatelessWidget {
  final String title;
  final Future<List<Product>> products;
  MyHomePage({Key key, this.title, this.products}) : super(key: key);

  // final items = Product.getProducts();
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Product Navigation")),
      body: Center(
        child: FutureBuilder<List<Product>>(
          future: products, builder: (context, snapshot) {
            if (snapshot.hasError) print(snapshot.error);
            return snapshot.hasData ? ProductBoxList(items:
snapshot.data)

            // return the ListView widget :
            Center(child: CircularProgressIndicator());
          },
        ),
      ),
    );
  }
}

```

```

    }
}
class ProductBoxList extends StatelessWidget {
  final List<Product> items;
  ProductBoxList({Key key, this.items});

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        return GestureDetector(
          child: ProductBox(item: items[index]),
          onTap: () {
            Navigator.push(
              context, MaterialPageRoute(
                builder: (context) => ProductPage(item:
items[index]),
              ),
            );
          },
        );
      },
    );
  }
}

class ProductPage extends StatelessWidget {
  ProductPage({Key key, this.item}) : super(key: key);
  final Product item;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(this.item.name)),
      body: Center(
        child: Container(
          padding: EdgeInsets.all(0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.start,
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
              Image.asset("assets/appimages/" +
this.item.image),
              Expanded(
                child: Container(
                  padding: EdgeInsets.all(5),
                  child: Column(
                    mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                    children: <Widget>[
                      Text(this.item.name, style:
                        TextStyle(fontWeight:
FontWeight.bold)),
                      Text(this.item.description),
                      Text("Price: " +
this.item.price.toString()),
                      RatingBox(),
                    ],
                  ),
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}

```

```

    ),
  ),
);
}
}
class RatingBox extends StatefulWidget {
  @override
  _RatingBoxState createState() => _RatingBoxState();
}
class _RatingBoxState extends State<RatingBox> {
  int _rating = 0;
  void _setRatingAsOne() {
    setState(() {
      _rating = 1;
    });
  }
  void _setRatingAsTwo() {
    setState(() {
      _rating = 2;
    });
  }
  void _setRatingAsThree() {
    setState(() {
      _rating = 3;
    });
  }
  Widget build(BuildContext context) {
    double _size = 20;
    print(_rating);
    return Row(
      mainAxisAlignment: MainAxisAlignment.end,
      crossAxisAlignment: CrossAxisAlignment.end,
      mainAxisAlignment: MainAxisAlignment.max,

      children: <Widget>[
        Container(
          padding: EdgeInsets.all(0),
          child: IconButton(
            icon: (
              _rating >= 1
              ? Icon(Icons.star, size: _size,)
              : Icon(Icons.star_border, size: _size,)
            ),
            color: Colors.red[500], onPressed: _setRatingAsOne,
iconSize: _size,
          ),
        Container(
          padding: EdgeInsets.all(0),
          child: IconButton(
            icon: (
              _rating >= 2
              ? Icon(Icons.star, size: _size,)
              : Icon(Icons.star_border, size: _size, )
            ),
            color: Colors.red[500],
            onPressed: _setRatingAsTwo,
            iconSize: _size,
          ),
        ),
      ],
    ),
  ),
);

```

```

        Container(
          padding: EdgeInsets.all(0),
          child: IconButton(
            icon: (
              _rating >= 3 ?
                Icon(Icons.star, size: _size,)
                : Icon(Icons.star_border, size: _size,)
            ),
            color: Colors.red[500],
            onPressed: _setRatingAsThree,
            iconSize: _size,
          ),
        ),
      ],
    );
  }
}

class ProductBox extends StatelessWidget {
  ProductBox({Key key, this.item}) : super(key: key);
  final Product item;

  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(2), height: 140,
      child: Card(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Image.asset("assets/appimages/" + this.item.image),
            Expanded(
              child: Container(
                padding: EdgeInsets.all(5),
                child: Column(
                  mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
                  children: <Widget>[
                    Text(this.item.name,
style: TextStyle(fontWeight: FontWeight.bold)),
                    Text(this.item.description),
                    Text("Price: " +
this.item.price.toString()),
                    RatingBox(),
                  ],
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}

```

Finally run the application to see the result. It will be same as our *Navigation* example except the data is from Internet instead of local, static data entered while coding the application.

Flutter - Database Concepts

Flutter provides many advanced packages to work with databases. The most important packages are –

- **sqlite** – Used to access and manipulate SQLite database, and
- **firebase_database** – Used to access and manipulate cloud hosted NoSQL database from Google.

In this chapter, let us discuss each of them in detail.

SQLite

SQLite database is the de-facto and standard SQL based embedded database engine. It is small and time-tested database engine. sqlite package provides a lot of functionality to work efficiently with SQLite database. It provides standard methods to manipulate SQLite database engine. The core functionality provided by sqlite package is as follows –

- Create / Open (openDatabase method) a SQLite database.
- Execute SQL statement (execute method) against SQLite database.
- Advanced query methods (query method) to reduce to code required to query and get information from SQLite database.

Let us create a product application to store and fetch product information from a standard SQLite database engine using sqlite package and understand the concept behind the SQLite database and sqlite package.

- Create a new Flutter application in Android studio, *product_sqlite_app*.
- Replace the default startup code (main.dart) with our *product_rest_app* code.
- Copy the assets folder from *product_nav_app* to *product_rest_app* and add assets inside the **pubspec.yaml* file.

```
flutter:
  assets:
    - assets/appimages/floppy.png
    - assets/appimages/iphone.png
    - assets/appimages/laptop.png
    - assets/appimages/pendrive.png
    - assets/appimages/pixel.png
    - assets/appimages/tablet.png
```

- Configure sqlite package in the pubspec.yaml file as shown below –

```
dependencies: sqlite: any
```

Use the latest version number of sqlite in place of any

- Configure path_provider package in the pubspec.yaml file as shown below –

```
dependencies: path_provider: any
```

- Here, path_provider package is used to get temporary folder path of the system and path of the application. Use the latest version number of *sqlite* in place of *any*.
- Android studio will alert that the pubspec.yaml is updated.

Pubspec has been edited Get dependencies Upgrade dependencies Ignore

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- In database, we need primary key, id as additional field along with Product properties like name, price, etc., So, add id property in the Product class. Also, add a new method, toMap to convert product object into Map object. fromMap and

toMap are used to serialize and de-serialize the Product object and it is used in database manipulation methods.

```
class Product {
  final int id;
  final String name;
  final String description;
  final int price;
  final String image;
  static final columns = ["id", "name", "description", "price",
"image"];
  Product(this.id, this.name, this.description, this.price,
this.image);
  factory Product.fromMap(Map<String, dynamic> data) {
    return Product(
      data['id'],
      data['name'],
      data['description'],
      data['price'],
      data['image'],
    );
  }
  Map<String, dynamic> toMap() => {
    "id": id,
    "name": name,
    "description": description,
    "price": price,
    "image": image
  };
}
```

- Create a new file, Database.dart in the lib folder to write SQLite related functionality.
- Import necessary import statement in Database.dart.

```
import 'dart:async';
import 'dart:io';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';
import 'Product.dart';
```

- Note the following points here –
 - **async** is used to write asynchronous methods.
 - **io** is used to access files and directories.
 - **path** is used to access dart core utility function related to file paths.
 - **path_provider** is used to get temporary and application path.
 - **sqflite** is used to manipulate SQLite database.
- Create a new class **SQLiteDbProvider**
- Declare a singleton based, static SQLiteDbProvider object as specified below –

```
class SQLiteDbProvider {
  SQLiteDbProvider._();
  static final SQLiteDbProvider db = SQLiteDbProvider._();
  static Database _database;
}
```

- SQLiteDBProvider object and its method can be accessed through the static db variable.

SQLiteDBProvider.db.<emthod>

- Create a method to get database (Future option) of type Future<Database>. Create product table and load initial data during the creation of the database itself.

```
Future<Database> get database async {
  if (_database != null)
    return _database;
  _database = await initDB();
  return _database;
}
initDB() async {
  Directory documentsDirectory = await
getApplicationDocumentsDirectory();
  String path = join(documentsDirectory.path, "ProductDB.db");
  return await openDatabase(
    path,
    version: 1,
    onOpen: (db) {},
    onCreate: (Database db, int version) async {
      await db.execute(
        "CREATE TABLE Product ("
        "id INTEGER PRIMARY KEY,"
        "name TEXT,"
        "description TEXT,"
        "price INTEGER,"
        "image TEXT" ")");
      await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [1, "iPhone", "iPhone is the stylist phone ever", 1000,
"iphone.png"]);
      await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [2, "Pixel", "Pixel is the most feature phone ever", 800,
"pixel.png"]);
      await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [3, "Laptop", "Laptop is most productive development
tool", 2000, "laptop.png"]);
      await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [4, "Tablet", "Laptop is most productive development
tool", 1500, "tablet.png"]);
      await db.execute(
        "INSERT INTO Product
('id', 'name', 'description', 'price', 'image')

```

```

        values (?, ?, ?, ?, ?)",
        [5, "Pendrive", "Pendrive is useful storage medium", 100,
"pendrive.png"]
    );
    await db.execute(
        "INSERT INTO Product
        ('id', 'name', 'description', 'price', 'image')
        values (?, ?, ?, ?, ?)",
        [6, "Floppy Drive", "Floppy drive is useful rescue storage
medium", 20, "floppy.png"]
    );
    }
    );
}

```

- Here, we have used the following methods –
 - **getApplicationDocumentsDirectory** – Returns application directory path
 - **join** – Used to create system specific path. We have used it to create database path.
 - **openDatabase** – Used to open a SQLite database
 - **onOpen** – Used to write code while opening a database
 - **onCreate** – Used to write code while a database is created for the first time
 - **db.execute** – Used to execute SQL queries. It accepts a query. If the query has placeholder (?), then it accepts values as list in the second argument.
- Write a method to get all products in the database –

```

Future<List<Product>> getAllProducts() async {
    final db = await database;
    List<Map>
    results = await db.query("Product", columns: Product.columns,
orderBy: "id ASC");

    List<Product> products = new List();
    results.forEach((result) {
        Product product = Product.fromMap(result);
        products.add(product);
    });
    return products;
}

```

- Here, we have done the following –
 - Used query method to fetch all the product information. query provides shortcut to query a table information without writing the entire query. query method will generate the proper query itself by using our input like columns, orderBy, etc.,
 - Used Product's fromMap method to get product details by looping the results object, which holds all the rows in the table.
- Write a method to get product specific to **id**

```

Future<Product> getProductById(int id) async {
    final db = await database;
    var result = await db.query("Product", where: "id = ", whereArgs:
[id]);
}

```



```

    return result.isNotEmpty ? Product.fromMap(result.first) : Null;
}

```

- Here, we have used where and whereArgs to apply filters.
- Create three methods - insert, update and delete method to insert, update and delete product from the database.

```

insert(Product product) async {
    final db = await database;
    var maxIdResult = await db.rawQuery(
        "SELECT MAX(id)+1 as last_inserted_id FROM Product");

    var id = maxIdResult.first["last_inserted_id"];
    var result = await db.rawQuery(
        "INSERT Into Product (id, name, description, price, image)"
        " VALUES (?, ?, ?, ?, ?)",
        [id, product.name, product.description, product.price,
product.image]
    );
    return result;
}

update(Product product) async {
    final db = await database;
    var result = await db.update("Product", product.toMap(),
        where: "id = ?", whereArgs: [product.id]); return result;
}

delete(int id) async {
    final db = await database;
    db.delete("Product", where: "id = ?", whereArgs: [id]);
}

```

- The final code of the Database.dart is as follows –

```

import 'dart:async';
import 'dart:io';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';
import 'Product.dart';

class SQLiteDatabaseProvider {
    SQLiteDatabaseProvider._();
    static final SQLiteDatabaseProvider db = SQLiteDatabaseProvider._();
    static Database _database;

    Future<Database> get database async {
        if (_database != null)
            return _database;
        _database = await initDB();
        return _database;
    }

    initDB() async {
        Directory documentsDirectory = await
        getApplicationDocumentsDirectory();
        String path = join(documentsDirectory.path, "ProductDB.db");
        return await openDatabase(
            path, version: 1,
            onOpen: (db) {},
            onCreate: (Database db, int version) async {
                await db.execute(
                    "CREATE TABLE Product ("

```

```

        "id INTEGER PRIMARY KEY,"
        "name TEXT,"
        "description TEXT,"
        "price INTEGER,"
        "image TEXT""")
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [1, "iPhone", "iPhone is the stylist phone ever", 1000,
"iphone.png"]
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [2, "Pixel", "Pixel is the most feature phone ever",
800, "pixel.png"]
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [3, "Laptop", "Laptop is most productive development
tool", 2000, "laptop.png"]
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [4, "Tablet", "Laptop is most productive development
tool", 1500, "tablet.png"]
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [5, "Pendrive", "Pendrive is useful storage medium",
100, "pendrive.png"]
    );
    await db.execute(
        "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
        values (?, ?, ?, ?, ?)",
        [6, "Floppy Drive", "Floppy drive is useful rescue
storage medium", 20, "floppy.png"]
    );
    }
    );
}

Future<List<Product>> getAllProducts() async {
    final db = await database;
    List<Map> results = await db.query(
        "Product", columns: Product.columns, orderBy: "id ASC"
    );
    List<Product> products = new List();
    results.forEach((result) {
        Product product = Product.fromMap(result);
        products.add(product);
    });
};

```

```

        return products;
    }
    Future<Product> getProductById(int id) async {
        final db = await database;
        var result = await db.query("Product", where: "id = ",
whereArgs: [id]);
        return result.isNotEmpty ? Product.fromMap(result.first) : Null;
    }
    insert(Product product) async {
        final db = await database;
        var maxIdResult = await db.rawQuery("SELECT MAX(id)+1 as
last_inserted_id FROM Product");
        var id = maxIdResult.first["last_inserted_id"];
        var result = await db.rawQuery(
            "INSERT Into Product (id, name, description, price, image)"
            " VALUES (?, ?, ?, ?, ?)",
            [id, product.name, product.description, product.price,
product.image]
        );
        return result;
    }
    update(Product product) async {
        final db = await database;
        var result = await db.update(
            "Product", product.toMap(), where: "id = ?", whereArgs:
[product.id]
        );
        return result;
    }
    delete(int id) async {
        final db = await database;
        db.delete("Product", where: "id = ?", whereArgs: [id]);
    }
}

```

- Change the main method to get the product information.

```

void main() {
    runApp(MyApp(products: SQLiteDbProvider.db.getAllProducts()));
}

```

- Here, we have used the getAllProducts method to fetch all products from the database.
- Run the application and see the results. It will be similar to previous example, *Accessing Product service API*, except the product information is stored and fetched from the local SQLite database.

Cloud Firestore

Firebase is a BaaS app development platform. It provides many feature to speed up the mobile application development like authentication service, cloud storage, etc., One of the main feature of Firebase is Cloud Firestore, a cloud based real time NoSQL database.

Flutter provides a special package, cloud_firestore to program with Cloud Firestore. Let us create an online product store in the Cloud Firestore and create a application to access the product store.

- Create a new Flutter application in Android studio, product_firebase_app.
- Replace the default startup code (main.dart) with our *product_rest_app* code.

- Copy Product.dart file from product_rest_app into the lib folder.

```
class Product {
  final String name;
  final String description;
  final int price;
  final String image;

  Product(this.name, this.description, this.price, this.image);
  factory Product.fromMap(Map<String, dynamic> json) {
    return Product(
      json['name'],
      json['description'],
      json['price'],
      json['image'],
    );
  }
}
```

- Copy the assets folder from product_rest_app to product_firebase_app and add assets inside the pubspec.yaml file.

```
flutter:
  assets:
    - assets/appimages/floppy.png
    - assets/appimages/iphone.png
    - assets/appimages/laptop.png
    - assets/appimages/pendrive.png
    - assets/appimages/pixel.png
    - assets/appimages/tablet.png
```

- Configure cloud_firestore package in the pubspec.yaml file as shown below –

```
dependencies: cloud_firestore: ^0.9.13+1
```

- Here, use the latest version of the cloud_firestore package.
- Android studio will alert that the pubspec.yaml is updated as shown here –

Pubspec has been edited Get dependencies Upgrade dependencies Ignore

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Create a project in the Firebase using the following steps –
 - Create a Firebase account by selecting Free plan at <https://firebase.google.com/pricing/>.
 - Once Firebase account is created, it will redirect to the project overview page. It list all the Firebase based project and provides an option to create a new project.
 - Click Add project and it will open a project creation page.
 - Enter products app db as project name and click Create project option.
 - Go to *Firebase console.
 - Click Project overview. It opens the project overview page.
 - Click android icon. It will open project setting specific to Android development.
 - Enter Android Package name, com.postparaprogramadores.flutterapp.product_firebase_app.

- Click Register App. It generates a project configuration file, google_service.json.
- Download google_service.json and then move it into the project's android/app directory. This file is the connection between our application and Firebase.
- Open android/app/build.gradle and include the following code –

```
apply plugin: 'com.google.gms.google-services'
```

- Open android/build.gradle and include the following configuration –

```
buildscript {
    repositories {
        // ...
    }
    dependencies {
        // ...
        classpath 'com.google.gms:google-services:3.2.1' // new
    }
}
```

Here, the plugin and class path are used for the purpose of reading google_service.json file.

- Open android/app/build.gradle and include the following code as well.

```
android {
    defaultConfig {
        ...
        multiDexEnabled true
    }
    ...
}
dependencies {
    ...
    compile 'com.android.support:multidex:1.0.3'
}
```

This dependency enables the android application to use multiple dex functionality.

- Follow the remaining steps in the Firebase Console or just skip it.
- Create a product store in the newly created project using the following steps –
 - Go to Firebase console.
 - Open the newly created project.
 - Click the Database option in the left menu.
 - Click Create database option.
 - Click Start in test mode and then Enable.
 - Click Add collection. Enter product as collection name and then click Next.
 - Enter the sample product information as shown in the image here –

Document parent path

/product

Document ID

pixel

Field	Type	Value
name	= string	Pixel
description	= string	Pixel is the most
price	= number	800
image	= string	pixel.png

Cancel Save

- Add additional product information using *Add document* options.
- Open main.dart file and import Cloud Firestore plugin file and remove http package.

```
import 'package:cloud_firestore/cloud_firestore.dart';
```

- Remove parseProducts and update fetchProducts to fetch products from Cloud Firestore instead of Product service API.

```
Stream<QuerySnapshot> fetchProducts() {
  return Firestore.instance.collection('product').snapshots(); }

```

- Here, Firestore.instance.collection method is used to access product collection available in the cloud store. Firestore.instance.collection provides many options to filter the collection to get the necessary documents. But, we have not applied any filter to get all product information.
- Cloud Firestore provides the collection through Dart Stream concept and so modify the products type in MyApp and MyHomePage widget from Future<list<Product>> to Stream<QuerySnapshot>.
- Change the build method of MyHomePage widget to use StreamBuilder instead of FutureBuilder.

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Product Navigation")),
    body: Center(
      child: StreamBuilder<QuerySnapshot>(
        stream: products, builder: (context, snapshot) {
          if (snapshot.hasError) print(snapshot.error);
          if (snapshot.hasData) {
            List<DocumentSnapshot>
              documents = snapshot.data.documents;

            List<Product>
              items = List<Product>();

            for (var i = 0; i < documents.length; i++) {
              DocumentSnapshot document = documents[i];
              items.add(Product.fromMap(document.data));
            }
            return ProductBoxList(items: items);
          } else {
            return Center(child: CircularProgressIndicator());
          }
        },
      ),
    ),
  );
}

```

- Here, we have fetched the product information as List<DocumentSnapshot> type. Since, our widget, ProductBoxList is not compatible with documents, we have converted the documents into List<Product> type and further used it.
- Finally, run the application and see the result. Since, we have used the same product information as that of *SQLite application* and changed the storage medium only, the resulting application looks identical to *SQLite application*.

Flutter - Internationalization

Nowadays, mobile applications are used by customers from different countries and as a result, applications are required to display the content in different languages. Enabling an application to work in multiple languages is called Internationalizing the application.

For an application to work in different languages, it should first find the current locale of the system in which the application is running and then need to show its content in that particular locale, and this process is called Localization.

Flutter framework provides three base classes for localization and extensive utility classes derived from base classes to localize an application.

The base classes are as follows –

- *Locale* – Locale is a class used to identify the user's language. For example, en-us identifies the American English and it can be created as.

```
Locale en_locale = Locale('en', 'US')
```

Here, the first argument is language code and the second argument is country code. Another example of creating *Argentina Spanish (es-ar)* locale is as follows –

```
Locale es_locale = Locale('es', 'AR')
```

- **Localizations** – Localizations is a generic widget used to set the Locale and the localized resources of its child.

```
class CustomLocalizations {
    CustomLocalizations(this.locale);
    final Locale locale;
    static CustomLocalizations of(BuildContext context) {
        return Localizations.of<CustomLocalizations>(context,
CustomLocalizations);
    }
    static Map<String, Map<String, String>> _resources = {
        'en': {
            'title': 'Demo',
            'message': 'Hello World'
        },
        'es': {
            'title': 'Manifestación',
            'message': 'Hola Mundo',
        },
    };
    String get title {
        return _resources[locale.languageCode]['title'];
    }
    String get message {
        return _resources[locale.languageCode]['message'];
    }
}
```

- Here, CustomLocalizations is a new custom class created specifically to get certain localized content (title and message) for the widget. of method uses the Localizations class to return new CustomLocalizations class.
- LocalizationsDelegate<T> – LocalizationsDelegate<T> is a factory class through which Localizations widget is loaded. It has three over-ridable methods –
 - isSupported – Accepts a locale and return whether the specified locale is supported or not.

```
@override
bool isSupported(Locale locale) => ['en',
'es'].contains(locale.languageCode);
```

Here, the delegate works for en and es locale only.

- load – Accepts a locale and start loading the resources for the specified locale.

```
@override
Future<CustomLocalizations> load(Locale locale) {
    return
SynchronousFuture<CustomLocalizations>(CustomLocalizations(locale));
}
```

Here, load method returns CustomLocalizations. The returned CustomLocalizations can be used to get values of title and message in both English and Spanish

- shouldReload – Specifies whether reloading of CustomLocalizations is necessary when its Localizations widget is rebuild.

```
@override
bool shouldReload(CustomLocalizationsDelegate old) => false;
```

- The complete code of CustomLocalizationDelegate is as follows –


```
class CustomLocalizationsDelegate extends
LocalizationsDelegate<CustomLocalizations> {
  const CustomLocalizationsDelegate();
  @override
  bool isSupported(Locale locale) => ['en',
'es'].contains(locale.languageCode);
  @override
  Future<CustomLocalizations> load(Locale locale) {
    return
SynchronousFuture<CustomLocalizations>(CustomLocalizations(locale));
  }
  @override bool shouldReload(CustomLocalizationsDelegate old) =>
false;
}
```

In general, Flutter applications are based on two root level widgets, MaterialApp or WidgetsApp. Flutter provides ready made localization for both widgets and they are MaterialLocalizations and WidgetsLocaliations. Further, Flutter also provides delegates to load MaterialApp and WidgetsLocaliations and they are GlobalMaterialLocalizations.delegate and GlobalWidgetsLocalizations.delegate respectively.

Let us create a simple internationalization enabled application to test and understand the concept.

- Create a new flutter application, flutter_localization_app.
- Flutter supports the internationalization using exclusive flutter package, flutter_localizations. The idea is to separate the localized content from the main SDK. Open the pubspec.yaml and add below code to enable the internationalization package –

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

- Android studio will display the following alert that the pubspec.yaml is updated.



- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Import flutter_localizations package in the main.dart as follows –

```
import 'package:flutter_localizations/flutter_localizations.dart';
import 'package:flutter/foundation.dart' show SynchronousFuture;
```

- Here, the purpose of SynchronousFuture is to load the custom localizations synchronously.
- Create a custom localizations and its corresponding delegate as specified below –

```
class CustomLocalizations {
  CustomLocalizations(this.locale);
  final Locale locale;
  static CustomLocalizations of(BuildContext context) {
    return Localizations.of<CustomLocalizations>(context,
CustomLocalizations);
  }
  static Map<String, Map<String, String>> _resources = {
    'en': {
```

```

        'title': 'Demo',
        'message': 'Hello World'
    },
    'es': {
        'title': 'Manifestación',
        'message': 'Hola Mundo',
    },
};
String get title {
    return _resources[locale.languageCode]['title'];
}
String get message {
    return _resources[locale.languageCode]['message'];
}
}
class CustomLocalizationsDelegate extends
LocalizationsDelegate<CustomLocalizations> {
    const CustomLocalizationsDelegate();

    @override
    bool isSupported(Locale locale) => ['en',
'es'].contains(locale.languageCode);

    @override
    Future<CustomLocalizations> load(Locale locale) {
        return
SynchronousFuture<CustomLocalizations>(CustomLocalizations(locale));
    }
    @override bool shouldReload(CustomLocalizationsDelegate old) =>
false;
}

```

- Here, CustomLocalizations is created to support localization for title and message in the application and CustomLocalizationsDelegate is used to load CustomLocalizations.
- Add delegates for MaterialApp, WidgetsApp and CustomLocalization using MaterialApp properties, localizationsDelegates and supportedLocales as specified below –

```

localizationsDelegates: [
    const CustomLocalizationsDelegate(),
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
],
supportedLocales: [
    const Locale('en', ''),
    const Locale('es', ''),
],

```

- Use CustomLocalizations method, of to get the localized value of title and message and use it in appropriate place as specified below –

```

class MyHomePage extends StatelessWidget {
    MyHomePage({Key key, this.title}) : super(key: key);
    final String title;
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text(CustomLocalizations.of(context)
.title), ),
            body: Center(

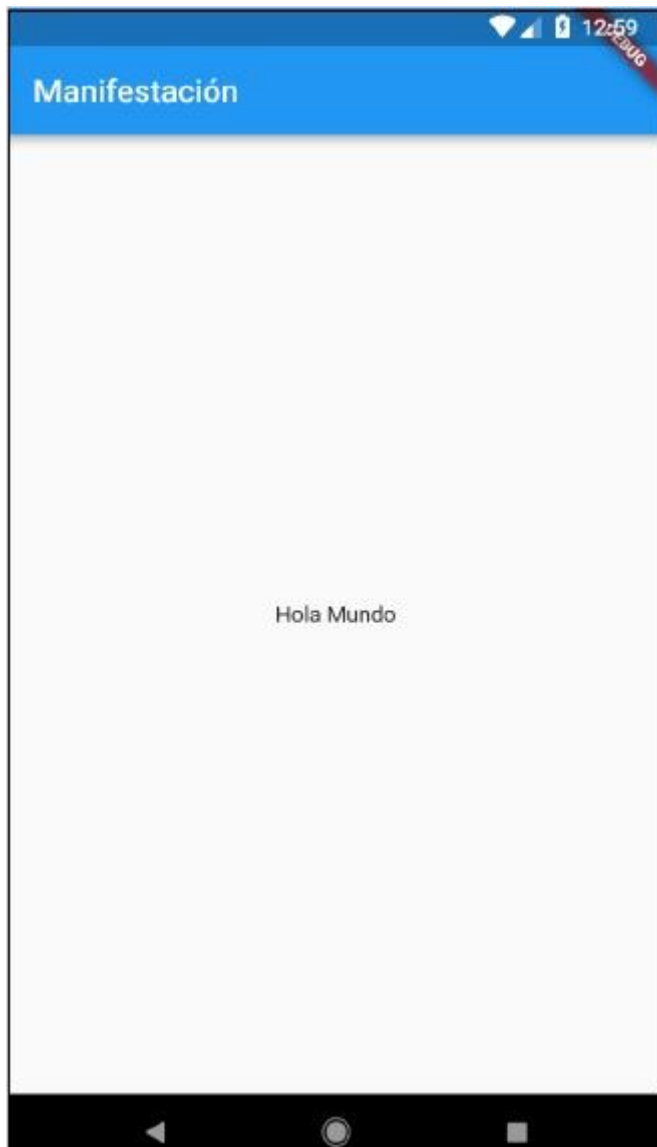
```

```

        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text( CustomLocalizations .of(context) .message, ),
          ],
        ),
      ),
    );
  }
}

```

- Here, we have modified the MyHomePage class from StatefulWidget to StatelessWidget for simplicity reason and used the CustomLocalizations to get title and message.
- Compile and run the application. The application will show its content in English.
- Close the application. Go to **Settings** → **System** → **Languages and Input** → **Languages***.
- Click Add a language option and select Spanish. This will install Spanish language and then list it as one of the option.
- Select Spanish and move it above English. This will set as Spanish as first language and everything will be changed to Spanish text.
- Now relaunch the internationalization application and you will see the title and message in Spanish language.
- We can revert the language to English by move the English option above Spanish option in the setting.
- The result of the application (in Spanish) is shown in the screenshot given below –



Using intl Package

Flutter provides intl package to further simplify the development of localized mobile application. intl package provides special methods and tools to semi-auto generate language specific messages.

Let us create a new localized application by using intl package and understand the concept.

- Create a new flutter application, flutter_intl_app.
- Open pubspec.yaml and add the package details.

```
dependencies:  
  flutter:  
    sdk: flutter  
  flutter_localizations:  
    sdk: flutter  
  intl: ^0.15.7  
  intl_translation: ^0.17.3
```

- Android studio will display the alert as shown below informing that the pubspec.yaml is updated.

Pubspec has been edited Get dependencies Upgrade dependencies Ignore

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Copy the main.dart from previous sample, flutter_internationalization_app.
- Import the intl package as shown below –

```
import 'package:intl/intl.dart';
```

- Update the CustomLocalization class as shown in the code given below –

```
class CustomLocalizations {
  static Future<CustomLocalizations> load(Locale locale) {
    final String name = locale.countryCode.isEmpty ?
    locale.languageCode : locale.toString();
    final String localeName = Intl.canonicalizedLocale(name);

    return initializeMessages(localeName).then((_) {
      Intl.defaultLocale = localeName;
      return CustomLocalizations();
    });
  }
  static CustomLocalizations of(BuildContext context) {
    return Localizations.of<CustomLocalizations>(context,
    CustomLocalizations);
  }
  String get title {
    return Intl.message(
      'Demo',
      name: 'title',
      desc: 'Title for the Demo application',
    );
  }
  String get message{
    return Intl.message(
      'Hello World',
      name: 'message',
      desc: 'Message for the Demo application',
    );
  }
}
class CustomLocalizationsDelegate extends
LocalizationsDelegate<CustomLocalizations> {
  const CustomLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) => ['en',
'es'].contains(locale.languageCode);
  @override
  Future<CustomLocalizations> load(Locale locale) {
    return CustomLocalizations.load(locale);
  }
  @override
  bool shouldReload(CustomLocalizationsDelegate old) => false;
}
```

- Here, we have used three methods from the intl package instead of custom methods. Otherwise, the concepts are same.

- Intl.canonicalizedLocale – Used to get correct locale name.
- Intl.defaultLocale – Used to set current locale
- Intl.message – Used to define new messages.
- import **l10n/messages_all.dart** file. We will generate this file shortly

```
import 'l10n/messages_all.dart';
```

- Now, create a folder, lib/l10n
- Open a command prompt and go to application root directory (where pubspec.yaml is available) and run the following command –

```
flutter packages pub run intl_translation:extract_to_arb --output-dir=lib/l10n lib/main.dart
```

- Here, the command will generate, intl_message.arb file, a template to create message in different locale. The content of the file is as follows –

```
{
  "@@last_modified": "2019-04-19T02:04:09.627551",
  "title": "Demo",
  "@title": {
    "description": "Title for the Demo application",
    "type": "text",
    "placeholders": {}
  },
  "message": "Hello World",
  "@message": {
    "description": "Message for the Demo application",
    "type": "text",
    "placeholders": {}
  }
}
```

- Copy intl_message.arb and create new file, intl_en.arb.
- Copy intl_message.arb and create new file, intl_es.arb and change the content to Spanish language as shown below –

```
{
  "@@last_modified": "2019-04-19T02:04:09.627551",
  "title": "Manifestación",
  "@title": {
    "description": "Title for the Demo application",
    "type": "text",
    "placeholders": {}
  },
  "message": "Hola Mundo",
  "@message": {
    "description": "Message for the Demo application",
    "type": "text",
    "placeholders": {}
  }
}
```

- Now, run the following command to create final message file, messages_all.dart.

```
flutter packages pub run intl_translation:generate_from_arb
--output-dir=lib\l10n --no-use-deferred-loading
lib\main.dart lib\l10n\intl_en.arb lib\l10n\intl_es.arb
```

- Compile and run the application. It will work similar to above application, flutter_localization_app.

Flutter - Testing

Testing is very important phase in the development life cycle of an application. It ensures that the application is of high quality. Testing requires careful planning and execution. It is also the most time consuming phase of the development.

Dart language and Flutter framework provides extensive support for the automated testing of an application.

Types of Testing

Generally, three types of testing processes are available to completely test an application. They are as follows –

Unit Testing

Unit testing is the easiest method to test an application. It is based on ensuring the correctness of a piece of code (a function, in general) or a method of a class. But, it does not reflect the real environment and subsequently, is the least option to find the bugs.

Widget Testing

Widget testing is based on ensuring the correctness of the widget creation, rendering and interaction with other widgets as expected. It goes one step further and provides near real-time environment to find more bugs.

Integration Testing

Integration testing involves both unit testing and widget testing along with external component of the application like database, web service, etc., It simulates or mocks the real environment to find nearly all bugs, but it is the most complicated process.

Flutter provides support for all types of testing. It provides extensive and exclusive support for Widget testing. In this chapter, we will discuss widget testing in detail.

Widget Testing

Flutter testing framework provides `testWidgets` method to test widgets. It accepts two arguments –

- Test description
- Test code

```
testWidgets('test description: find a widget', '<test code>');
```

Steps Involved

Widget Testing involves three distinct steps –

- Render the widget in the testing environment.

- WidgetTester is the class provided by Flutter testing framework to build and renders the widget. pumpWidget method of the WidgetTester class accepts any widget and renders it in the testing environment.

```
testWidgets('finds a specific instance', (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(
    home: Scaffold(
      body: Text('Hello'),
    ),
  ));
});
```

- Finding the widget, which we need to test.
 - Flutter framework provides many options to find the widgets rendered in the testing environment and they are generally called Finders. The most frequently used finders are find.text, find.byKey and find.byWidget.
 - find.text finds the widget that contains the specified text.

```
find.text('Hello')
```

- find.byKey find the widget by its specific key.

```
find.byKey('home')
```

- find.byWidget find the widget by its instance variable.

```
find.byWidget(homeWidget)
```

- Ensuring the widget works as expected.
- Flutter framework provides many options to match the widget with the expected widget and they are normally called *Matchers*. We can use the expect method provided by the testing framework to match the widget, which we found in the second step with our expected widget by choosing any of the matchers. Some of the important matchers are as follows.

- findsOneWidget – verifies a single widget is found.

```
expect(find.text('Hello'), findsOneWidget);
```

- findsNothing – verifies no widgets are found

```
expect(find.text('Hello World'), findsNothing);
```

- findsWidgets – verifies more than a single widget is found.

```
expect(find.text('Save'), findsWidgets);
```

- findsNWidgets – verifies N number of widgets are found.

```
expect(find.text('Save'), findsNWidgets(2));
```

The complete test code is as follows –

```
testWidgets('finds hello widget', (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(
    home: Scaffold(
      body: Text('Hello'),
    ),
  ));
  expect(find.text('Hello'), findsOneWidget);
});
```

Here, we rendered a MaterialApp widget with text Hello using Text widget in its body. Then, we used find.text to find the widget and then matched it using findsOneWidget.

Working Example

Let us create a simple flutter application and write a widget test to understand better the steps involved and the concept.

- Create a new flutter application, flutter_test_app in Android studio.
- Open widget_test.dart in test folder. It has a sample testing code as given below –

```
testWidgets('Counter increments smoke test', (WidgetTester tester)
  async {
    // Build our app and trigger a frame.
    await tester.pumpWidget(MyApp());

    // Verify that our counter starts at 0.
    expect(find.text('0'), findsOneWidget);
    expect(find.text('1'), findsNothing);

    // Tap the '+' icon and trigger a frame.
    await tester.tap(find.byIcon(Icons.add));
    await tester.pump();

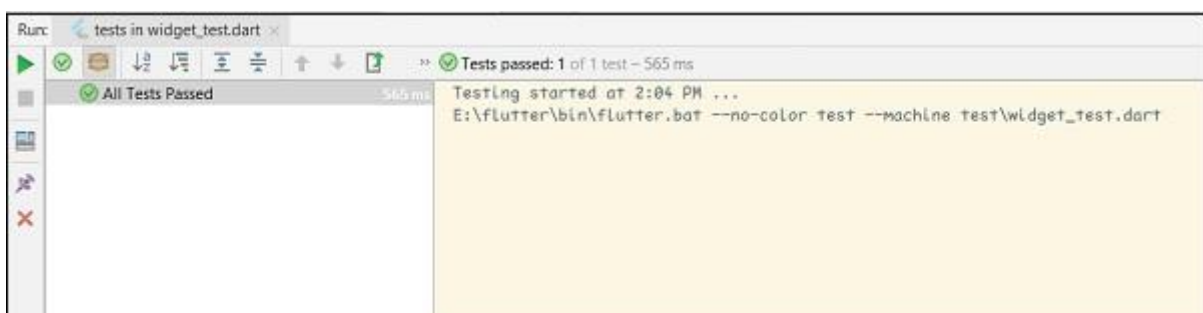
    // Verify that our counter has incremented.
    expect(find.text('0'), findsNothing);
    expect(find.text('1'), findsOneWidget);
  });
```

- Here, the test code does the following functionalities –
 - Renders MyApp widget using tester.pumpWidget.
 - Ensures that the counter is initially zero using findsOneWidget and findsNothing matchers.
 - Finds the counter increment button using find.byIcon method.
 - Taps the counter increment button using tester.tap method.
 - Ensures that the counter is increased using findsOneWidget and findsNothing matchers.
- Let us again tap the counter increment button and then check whether the counter is increased to two.

```
await tester.tap(find.byIcon(Icons.add));
await tester.pump();
```

```
expect(find.text('2'), findsOneWidget);
```

- Click Run menu.
- Click tests in widget_test.dart option. This will run the test and report the result in the result window.



Flutter - Deployment

This chapter explains how to deploy Flutter application in both Android and iOS platforms.

Android Application

- Change the application name using android:label entry in android manifest file. Android app manifest file, AndroidManifest.xml is located in <app dir>/android/app/src/main. It contains entire details about an android application. We can set the application name using android:label entry.
- Change launcher icon using android:icon entry in manifest file.
- Sign the app using standard option as necessary.
- Enable Proguard and Obfuscation using standard option, if necessary.
- Create a release APK file by running below command –

```
cd /path/to/my/application
flutter build apk
```

- You can see an output as shown below –

```
Initializing gradle... 8.6s
Resolving dependencies... 19.9s
Calling mockable JAR artifact transform to create file:
/Users/.gradle/caches/transforms-1/files-1.1/android.jar/
c30932f130afbf3fd90c131ef9069a0b/android.jar with input
/Users/Library/Android/sdk/platforms/android-28/android.jar
Running Gradle task 'assembleRelease'...
Running Gradle task 'assembleRelease'...
Done 85.7s
Built build/app/outputs/apk/release/app-release.apk (4.8MB).
```

- Install the APK on a device using the following command –

```
flutter install
```

- Publish the application into Google Playstore by creating an appbundle and push it into playstore using standard methods.

```
flutter build appbundle
```

iOS Application

- Register the iOS application in *App Store Connect* using standard method. Save the **=Bundle ID** used while registering the application.
- Update Display name in the XCode project setting to set the application name.
- Update Bundle Identifier in the XCode project setting to set the bundle id, which we used in step 1.
- Code sign as necessary using standard method.
- Add a new app icon as necessary using standard method.
- Generate IPA file using the following command –

```
flutter build ios
```

- Now, you can see the following output –

```
Building com.example.MyApp for device (ios-release)...
Automatically signing iOS for device deployment
using specified development team in Xcode project:
Running Xcode build... 23.5s
.....
```

- Test the application by pushing the application, IPA file into TestFlight using standard method.
- Finally, push the application into *App Store* using standard method.

Flutter - Development Tools

This chapter explains about Flutter development tools in detail. The first stable release of the cross-platform development toolkit was released on December 4th, 2018, Flutter 1.0. Well, Google is continuously working on the improvements and strengthening the Flutter framework with different development tools.

Widget Sets

Google updated for Material and Cupertino widget sets to provide pixel-perfect quality in the components design. The upcoming version of flutter 1.2 will be designed to support desktop keyboard events and mouse hover support.

Flutter Development with Visual Studio Code

Visual Studio Code supports flutter development and provides extensive shortcuts for fast and efficient development. Some of the key features provided by Visual Studio Code for flutter development are listed below –

- Code assist - When you want to check for options, you can use **Ctrl+Space** to get a list of code completion options.
- Quick fix - **Ctrl+.** is quick fix tool to help in fixing the code.
- Shortcuts while Coding.
- Provides detailed documentation in comments.
- Debugging shortcuts.
- Hot restarts.

Dart DevTools

We can use Android Studio or Visual Studio Code, or any other IDE to write our code and install plugins. Google's development team has been working on yet another development tool called Dart DevTools It is a web-based programming suite. It supports both Android and iOS platforms. It is based on time line view so developers can easily analyze their applications.

Install DevTools

To install DevTools run the following command in your console –

```
flutter packages pub global activate devtools
```

Now you can see the following output –

```
Resolving dependencies...
+ args 1.5.1
+ async 2.2.0
+ charcode 1.1.2
+ codemirror 0.5.3+5.44.0
+ collection 1.14.11
+ convert 2.1.1
+ devtools 0.0.16
+ devtools_server 0.0.2
+ http 0.12.0+2
+ http_parser 3.1.3
+ intl 0.15.8
+ js 0.6.1+1
+ meta 1.1.7
+ mime 0.9.6+2
.....
.....
Installed executable devtools.
Activated devtools 0.0.16.
```

Run Server

You can run the DevTools server using the following command –

```
flutter packages pub global run devtools
```

Now, you will get a response similar to this,

```
Serving DevTools at http://127.0.0.1:9100
```

Start Your Application

Go to your application, open simulator and run using the following command –

```
flutter run --observatory-port=9200
```

Now, you are connected to DevTools.

Start DevTools in Browser

Now access the below url in the browser, to start DevTools –

```
http://localhost:9100/?port=9200
```

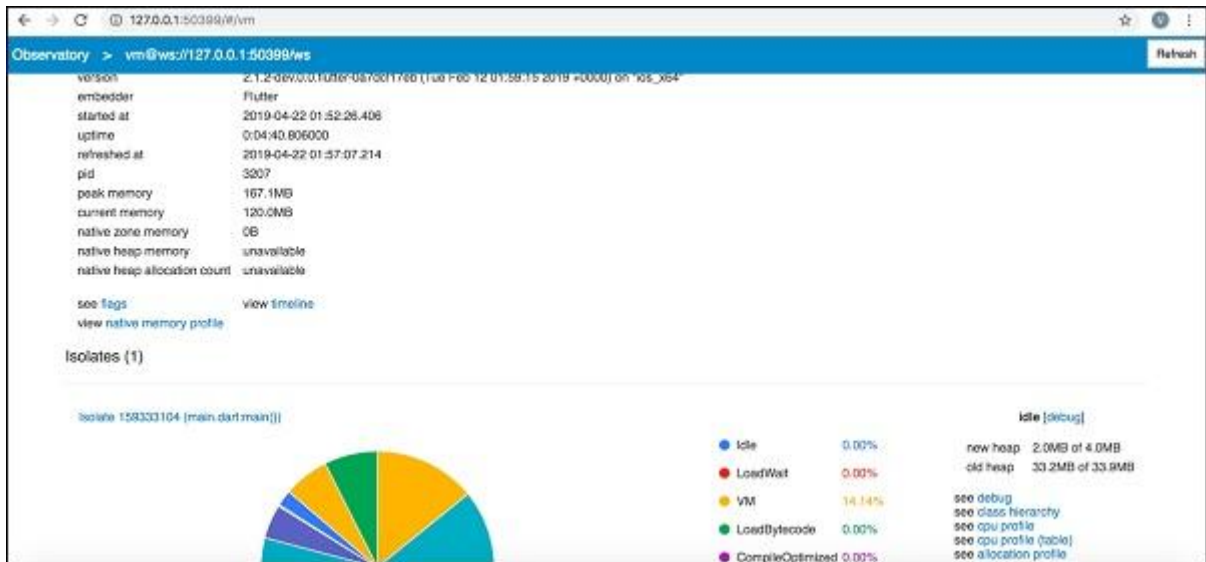
You will get a response as shown below –


```

-Assembling Flutter resources... 3.6s
Compiling, linking and signing... 6.8s
Xcode build done. 14.2s
2,904ms (!)
To hot reload changes while running, press "r". To hot restart (and
rebuild state), press "R".
An Observatory debugger and profiler on iPhone X is available at:
http://127.0.0.1:50399/
For a more detailed help message, press "h". To detach, press "d"; to
quit, press "q".

```

Now go to the url, <http://127.0.0.1:50399/> you could see the following result –



Flutter - Writing Advanced Applications

In this chapter, we are going to learn how to write a full fledged mobile application, `expense_calculator`. The purpose of the `expense_calculator` is to store our expense information. The complete feature of the application is as follows –

- Expense list.
- Form to enter new expenses.
- Option to edit / delete the existing expenses.
- Total expenses at any instance.

We are going to program the `expense_calculator` application using below mentioned advanced features of Flutter framework.

- Advanced use of ListView to show the expense list.
- Form programming.
- SQLite database programming to store our expenses.
- `scoped_model` state management to simplify our programming.

Let us start programming the **`expense_calculator`** application.

- Create a new Flutter application, `expense_calculator` in Android studio.
- Open `pubspec.yaml` and add package dependencies.

```

dependencies:
  flutter:

```

```
    sdk: flutter
  sqflite: ^1.1.0
  path_provider: ^0.5.0+1
  scoped_model: ^1.0.1
  intl: any
```

- Observe these points here –
 - sqflite is used for SQLite database programming.
 - path_provider is used to get system specific application path.
 - scoped_model is used for state management.
 - intl is used for date formatting.
- Android studio will display the following alert that the pubspec.yaml is updated.

Pubspec has been edited

Get dependencies Upgrade dependencies Ignore ⚙

- Click Get dependencies option. Android studio will get the package from Internet and properly configure it for the application.
- Remove the existing code in main.dart.
- Add new file, Expense.dart to create Expense class. Expense class will have the below properties and methods.
 - **property: id** – Unique id to represent an expense entry in SQLite database.
 - **property: amount** – Amount spent.
 - **property: date** – Date when the amount is spent.
 - **property: category** – Category represents the area in which the amount is spent. e.g Food, Travel, etc.,
 - **formattedDate** – Used to format the date property
 - **fromMap** – Used to map the field from database table to the property in the expense object and to create a new expense object.

```
factory Expense.fromMap(Map<String, dynamic> data) {
  return Expense(
    data['id'],
    data['amount'],
    DateTime.parse(data['date']),
    data['category']
  );
}
```

- **toMap** – Used to convert the expense object to Dart Map, which can be further used in database programming

```
Map<String, dynamic> toMap() => {
  "id" : id,
  "amount" : amount,
  "date" : date.toString(),
  "category" : category,
};
```

- **columns** – Static variable used to represent the database field.
- Enter and save the following code into the Expense.dart file.

```
import 'package:intl/intl.dart'; class Expense {
```

```

final int id;
final double amount;
final DateTime date;
final String category;
String get formattedDate {
    var formatter = new DateFormat('yyyy-MM-dd');
    return formatter.format(this.date);
}
static final columns = ['id', 'amount', 'date', 'category'];
Expense(this.id, this.amount, this.date, this.category);
factory Expense.fromMap(Map<String, dynamic> data) {
    return Expense(
        data['id'],
        data['amount'],
        DateTime.parse(data['date']), data['category']
    );
}
Map<String, dynamic> toMap() => {
    "id" : id,
    "amount" : amount,
    "date" : date.toString(),
    "category" : category,
};
}

```

- The above code is simple and self explanatory.
- Add new file, Database.dart to create SQLiteDatabaseProvider class. The purpose of the SQLiteDatabaseProvider class is as follows –
 - Get all expenses available in the database using getAllExpenses method. It will be used to list all the user's expense information.

```

Future<List<Expense>> getAllExpenses() async {
    final db = await database;

    List<Map> results = await db.query(
        "Expense", columns: Expense.columns, orderBy: "date DESC"
    );
    List<Expense> expenses = new List();
    results.forEach((result) {
        Expense expense = Expense.fromMap(result);
        expenses.add(expense);
    });
    return expenses;
}

```

- Get a specific expense information based on expense identity available in the database using getExpenseById method. It will be used to show the particular expense information to the user.

```

Future<Expense> getExpenseById(int id) async {
    final db = await database;
    var result = await db.query("Expense", where: "id = ", whereArgs:
[id]);

    return result.isNotEmpty ?
Expense.fromMap(result.first) : Null;
}

```

- Get the total expenses of the user using getTotalExpense method. It will be used to show the current total expense to the user.


```
Future<double> getTotalExpense() async {
  final db = await database;
  List<Map> list = await db.rawQuery(
    "Select SUM(amount) as amount from expense"
  );
  return list.isNotEmpty ? list[0]["amount"] : Null;
}
```

- Add new expense information into the database using insert method. It will be used to add new expense entry into the application by the user.

```
Future<Expense> insert(Expense expense) async {
  final db = await database;
  var maxIdResult = await db.rawQuery(
    "SELECT MAX(id)+1 as last_inserted_id FROM Expense"
  );
  var id = maxIdResult.first["last_inserted_id"];
  var result = await db.rawInsert(
    "INSERT Into Expense (id, amount, date, category)"
    " VALUES (?, ?, ?, ?)", [
      id, expense.amount, expense.date.toString(), expense.category
    ]
  );
  return Expense(id, expense.amount, expense.date, expense.category);
}
```

- Update existing expense information using update method. It will be used to edit and update existing expense entry available in the system by the user.

```
update(Expense product) async {
  final db = await database;

  var result = await db.update("Expense", product.toMap(),
    where: "id = ?", whereArgs: [product.id]);
  return result;
}
```

- Delete existing expense information using delete method. It will be used to remove the existing expense entry available in the system by the user.

```
delete(int id) async {
  final db = await database;
  db.delete("Expense", where: "id = ?", whereArgs: [id]);
}
```

- The complete code of the SQLiteDbProvider class is as follows –

```
import 'dart:async';
import 'dart:io';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';
import 'Expense.dart';

class SQLiteDbProvider {
  SQLiteDbProvider._();
  static final SQLiteDbProvider db = SQLiteDbProvider._();

  static Database _database; Future<Database> get database async {
    if (_database != null)
      return _database;
  }
}
```

```

        _database = await initDB();
        return _database;
    }
    initDB() async {
        Directory documentsDirectory = await
getApplicationDocumentsDirectory();
        String path = join(documentsDirectory.path, "ExpenseDB2.db");
        return await openDatabase(
            path, version: 1, onOpen: (db) {}, onCreate: (Database db, int
version) async {
                await db.execute(
                    "CREATE TABLE Expense (
                        \"id INTEGER PRIMARY KEY,\" \"amount REAL,\" \"date
TEXT,\" \"category TEXT\"\"
                    )
                ");
                await db.execute(
                    "INSERT INTO Expense ('id', 'amount', 'date',
'category')
                    values (?, ?, ?, ?)", [1, 1000, '2019-04-01 10:00:00',
"Food"]
                );
                /*await db.execute(
                    "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
                    values (?, ?, ?, ?, ?)", [
                        2, "Pixel", "Pixel is the most feature phone ever",
800, "pixel.png"
                    ]
                );
                await db.execute(
                    "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
                    values (?, ?, ?, ?, ?)", [
                        3, "Laptop", "Laptop is most productive development
tool", 2000, "laptop.png"
                    ]
                );
                await db.execute(
                    "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
                    values (?, ?, ?, ?, ?)", [
                        4, "Tablet", "Laptop is most productive development
tool", 1500, "tablet.png"
                    ]
                );
                await db.execute(
                    "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
                    values (?, ?, ?, ?, ?)", [
                        5, "Pendrive", "iPhone is the stylist phone ever",
100, "pendrive.png"
                    ]
                );
                await db.execute(
                    "INSERT INTO Product ('id', 'name', 'description',
'price', 'image')
                    values (?, ?, ?, ?, ?)", [
                        6, "Floppy Drive", "iPhone is the stylist phone
ever", 20, "floppy.png"
                    ]
                );
            }
        );
    }
}

```

```

        ); */
    }
}

Future<List<Expense>> getAllExpenses() async {
    final db = await database;
    List<Map>
    results = await db.query(
        "Expense", columns: Expense.columns, orderBy: "date DESC"
    );
    List<Expense> expenses = new List();
    results.forEach((result) {
        Expense expense = Expense.fromMap(result);
        expenses.add(expense);
    });
    return expenses;
}

Future<Expense> getExpenseById(int id) async {
    final db = await database;
    var result = await db.query("Expense", where: "id = ",
whereArgs: [id]);
    return result.isNotEmpty ? Expense.fromMap(result.first) : Null;
}

Future<double> getTotalExpense() async {
    final db = await database;
    List<Map> list = await db.rawQuery(
        "Select SUM(amount) as amount from expense"
    );
    return list.isNotEmpty ? list[0]["amount"] : Null;
}

Future<Expense> insert(Expense expense) async {
    final db = await database;
    var maxIdResult = await db.rawQuery(
        "SELECT MAX(id)+1 as last_inserted_id FROM Expense"
    );
    var id = maxIdResult.first["last_inserted_id"];
    var result = await db.rawQuery(
        "INSERT Into Expense (id, amount, date, category)"
        " VALUES (?, ?, ?, ?)", [
            id, expense.amount, expense.date.toString(),
expense.category
        ]
    );
    return Expense(id, expense.amount, expense.date,
expense.category);
}

update(Expense product) async {
    final db = await database;
    var result = await db.update(
        "Expense", product.toMap(), where: "id = ?", whereArgs:
[product.id]
    );
    return result;
}

delete(int id) async {
    final db = await database;
    db.delete("Expense", where: "id = ?", whereArgs: [id]);
}
}

```

- Here,

- database is the property to get the SQLiteDatabaseProvider object.
- initDB is a method used to select and open the SQLite database.
- Create a new file, ExpenseListModel.dart to create ExpenseListModel. The purpose of the model is to hold the complete information of the user expenses in the memory and updating the user interface of the application whenever user's expense changes in the memory. It is based on Model class from scoped_model package. It has the following properties and methods –
 - _items – private list of expenses.
 - items – getter for _items as UnmodifiableListView<Expense> to prevent unexpected or accidental changes to the list.
 - totalExpense – getter for Total expenses based on the items variable.

```
double get totalExpense {
  double amount = 0.0;
  for(var i = 0; i < _items.length; i++) {
    amount = amount + _items[i].amount;
  }
  return amount;
}
```

- load – Used to load the complete expenses from database and into the _items variable. It also calls notifyListeners to update the UI.

```
void load() {
  Future<List<Expense>>
  list = SQLiteDatabaseProvider.db.getAllExpenses();
  list.then((dbItems) {
    for(var i = 0; i < dbItems.length; i++) {
      _items.add(dbItems[i]);
    } notifyListeners();
  });
}
```

- byId – Used to get a particular expenses from _items variable.

```
Expense byId(int id) {
  for(var i = 0; i < _items.length; i++) {
    if(_items[i].id == id) {
      return _items[i];
    }
  }
  return null;
}
```

- add – Used to add a new expense item into the _items variable as well as into the database. It also calls notifyListeners to update the UI.

```
void add(Expense item) {
  SQLiteDatabaseProvider.db.insert(item).then((val) {
    _items.add(val); notifyListeners();
  });
}
```

- Update – Used to Update expense item into the _items variable as well as into the database. It also calls notifyListeners to update the UI.

```
void update(Expense item) {
  bool found = false;
```

```

    for(var i = 0; i < _items.length; i++) {
      if(_items[i].id == item.id) {
        _items[i] = item;
        found = true;
        SQLiteDatabaseProvider.db.update(item); break;
      }
    }
    if(found) notifyListeners();
  }
}

```

- delete – Used to remove an existing expense item in the _items variable as well as from the database. It also calls notifyListeners to update the UI.

```

void delete(Expense item) {
  bool found = false;
  for(var i = 0; i < _items.length; i++) {
    if(_items[i].id == item.id) {
      found = true;
      SQLiteDatabaseProvider.db.delete(item.id);
      _items.removeAt(i); break;
    }
  }
  if(found) notifyListeners();
}

```

- The complete code of the ExpenseListModel class is as follows –

```

import 'dart:collection';
import 'package:scoped_model/scoped_model.dart';
import 'Expense.dart';
import 'Database.dart';

class ExpenseListModel extends Model {
  ExpenseListModel() {
    this.load();
  }
  final List<Expense> _items = [];
  UnmodifiableListView<Expense> get items =>
    UnmodifiableListView(_items);

  /*Future<double> get totalExpense {
    return SQLiteDatabaseProvider.db.getTotalExpense();
  }*/

  double get totalExpense {
    double amount = 0.0;
    for(var i = 0; i < _items.length; i++) {
      amount = amount + _items[i].amount;
    }
    return amount;
  }
  void load() {
    Future<List<Expense>> list =
    SQLiteDatabaseProvider.db.getAllExpenses();
    list.then( (dbItems) {
      for(var i = 0; i < dbItems.length; i++) {
        _items.add(dbItems[i]);
      }
      notifyListeners();
    });
  }
  Expense byId(int id) {

```

```

        for(var i = 0; i < _items.length; i++) {
            if(_items[i].id == id) {
                return _items[i];
            }
        }
        return null;
    }
}

void add(Expense item) {
    SQLiteDatabaseProvider.db.insert(item).then((val) {
        _items.add(val);
        notifyListeners();
    });
}

void update(Expense item) {
    bool found = false;
    for(var i = 0; i < _items.length; i++) {
        if(_items[i].id == item.id) {
            _items[i] = item;
            found = true;
            SQLiteDatabaseProvider.db.update(item);
            break;
        }
    }
    if(found) notifyListeners();
}

void delete(Expense item) {
    bool found = false;
    for(var i = 0; i < _items.length; i++) {
        if(_items[i].id == item.id) {
            found = true;
            SQLiteDatabaseProvider.db.delete(item.id);
            _items.removeAt(i); break;
        }
    }
    if(found) notifyListeners();
}
}
}

```

- Open main.dart file. Import the classes as specified below –

```

import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';
import 'ExpenseListModel.dart';
import 'Expense.dart';

```

- Add main function and call runApp by passing ScopedModel<ExpenseListModel> widget.

```

void main() {
    final expenses = ExpenseListModel();
    runApp(
        ScopedModel<ExpenseListModel>(model: expenses, child: MyApp(),)
    );
}

```

- Here,
 - expenses object loads all the user expenses information from the database. Also, when the application is opened for the first time, it will create the required database with proper tables.
 - ScopedModel provides the expense information during the whole life cycle of the application and ensures the maintenance of state of the application

at any instance. It enables us to use StatelessWidget instead of StatefulWidget.

- Create a simple MyApp using MaterialApp widget.

```
class MyApp extends StatelessWidget {  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Expense',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(title: 'Expense calculator'),  
    );  
  }  
}
```

- Create MyHomePage widget to display all the user's expense information along with total expenses at the top. Floating button at the bottom right corner will be used to add new expenses.

```
class MyHomePage extends StatelessWidget {  
  MyHomePage({Key key, this.title}) : super(key: key);  
  final String title;  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(this.title),  
      ),  
      body: ScopedModelDescendant<ExpenseListModel>(  
        builder: (context, child, expenses) {  
          return ListView.separated(  
            itemCount: expenses.items == null ? 1  
              : expenses.items.length + 1,  
            itemBuilder: (context, index) {  
              if (index == 0) {  
                return ListTile(  
                  title: Text("Total expenses: "  
                    + expenses.totalExpense.toString(),  
                  style: TextStyle(fontSize: 24,  
                    fontWeight: FontWeight.bold),  
                );  
              } else {  
                index = index - 1;  
                return Dismissible(  
                  key:  
                    Key(expenses.items[index].id.toString()),  
                  onDismissed: (direction) {  
                    expenses.delete(expenses.items[index]);  
                    Scaffold.of(context).showSnackBar(  
                      SnackBar(  
                        content: Text(  
                          "Item with id, "  
                          +  
expenses.items[index].id.toString() +  
                          " is dismissed"  
                        ),  
                      ),  
                    );  
                  },  
                );  
              }  
            },  
          );  
        },  
      ),  
    );  
  }  
}
```

```

        },
        child: ListTile( onTap: () {
            Navigator.push(
                context, MaterialPageRoute(
                    builder: (context) => FormPage(
                        id: expenses.items[index].id,
                        expenses: expenses,
                    )
                )
            );
        },
        leading: Icon(Icons.monetization_on),
        trailing: Icon(Icons.keyboard_arrow_right),
        title: Text(expenses.items[index].category
+ ": " +
            expenses.items[index].amount.toString() +
            " \nspent on " +
expenses.items[index].formattedDate,
            style: TextStyle(fontSize: 18, fontStyle:
FontStyle.italic),))
        );
    },
    separatorBuilder: (context, index) {
        return Divider();
    },
),
),
floatingActionButton:
ScopedModelDescendant<ExpenseListModel>(
    builder: (context, child, expenses) {
        return FloatingActionButton( onPressed: () {
            Navigator.push(
                context, MaterialPageRoute(
                    builder: (context) =>
ScopedModelDescendant<ExpenseListModel>(
                    builder: (context, child, expenses) {
                        return FormPage( id: 0, expenses:
expenses, );
                    }
                )
            );
            // expenses.add(new Expense(
            // 2, 1000, DateTime.parse('2019-04-01
11:00:00'), 'Food')
            );
            // print(expenses.items.length);
        },
        tooltip: 'Increment', child: Icon(Icons.add), );
    }
);
}
}

```

- Here,
 - ScopedModelDescendant is used to pass the expense model into the ListView and FloatingActionButton widget.

- ListView.separated and ListTile widget is used to list the expense information.
 - Dismissible widget is used to delete the expense entry using swipe gesture.
 - Navigator is used to open edit interface of an expense entry. It can be activated by tapping an expense entry.
- Create a FormPage widget. The purpose of the FormPage widget is to add or update an expense entry. It handles expense entry validation as well.

```
class FormPage extends StatefulWidget {
  FormPage({Key key, this.id, this.expenses}) : super(key: key);
  final int id;
  final ExpenseListModel expenses;

  @override _FormPageState createState() => _FormPageState(id: id,
    expenses: expenses);
}
class _FormPageState extends State<FormPage> {
  _FormPageState({Key key, this.id, this.expenses});

  final int id;
  final ExpenseListModel expenses;
  final scaffoldKey = GlobalKey<ScaffoldState>();
  final formKey = GlobalKey<FormState>();

  double _amount;
  DateTime _date;
  String _category;

  void _submit() {
    final form = formKey.currentState;
    if (form.validate()) {
      form.save();
      if (this.id == 0) expenses.add(Expense(0, _amount, _date,
        _category));
      else expenses.update(Expense(this.id, _amount, _date,
        _category));
      Navigator.pop(context);
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      key: scaffoldKey, appBar: AppBar(
        title: Text('Enter expense details'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: formKey, child: Column(
            children: [
              TextFormField(
                style: TextStyle(fontSize: 22),
                decoration: const InputDecoration(
                  icon: const Icon(Icons.monetization_on),
                  labelText: 'Amount',
                  labelStyle: TextStyle(fontSize: 18)
                ),
                validator: (val) {
                  Pattern pattern = r'^[1-9]\d*(\.\d+)?$';
```

```

        Regex regex = new Regex(pattern);
        if (!regex.IsMatch(val))
            return 'Enter a valid number'; else return
null;

    },
    initialValue: id == 0
    ? '' : expensesById(id).amount.ToString(),
    onSave: (val) => _amount = double.Parse(val),
),
TextField(
    style: TextStyle(fontSize: 22),
    decoration: const InputDecoration(
        icon: const Icon(Icons.calendar_today),
        hintText: 'Enter date',
        labelText: 'Date',
        labelStyle: TextStyle(fontSize: 18),
    ),
    validator: (val) {
        Pattern pattern = r'^((?:19|20)\d\d)[- /.]
(0[1-9]|1[012])[- /.](0[1-9]|[12][0-
9]|13[01])$';

        Regex regex = new Regex(pattern);
        if (!regex.IsMatch(val))
            return 'Enter a valid date';
        else return null;
    },
    onSave: (val) => _date = DateTime.Parse(val),
    initialValue: id == 0
    ? '' : expensesById(id).formattedDate,
    keyboardType: TextInputType.datetime,
),
TextField(
    style: TextStyle(fontSize: 22),
    decoration: const InputDecoration(
        icon: const Icon(Icons.category),
        labelText: 'Category',
        labelStyle: TextStyle(fontSize: 18)
    ),
    onSave: (val) => _category = val,
    initialValue: id == 0 ? ''
    : expensesById(id).category.ToString(),
),
RaisedButton(
    onPressed: _submit,
    child: new Text('Submit'),
),
),
),
),
),
);
}
}

```

- Here,
 - TextFormField is used to create form entry.
 - validator property of TextFormField is used to validate the form element along with RegEx patterns.

- `_submit` function is used along with `expenses` object to add or update the expenses into the database.

- The complete code of the `main.dart` file is as follows –

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';
import 'ExpenseListModel.dart';
import 'Expense.dart';

void main() {
  final expenses = ExpenseListModel();
  runApp(
    ScopedModel<ExpenseListModel>(
      model: expenses, child: MyApp(),
    )
  );
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Expense',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Expense calculator'),
    );
  }
}

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: ScopedModelDescendant<ExpenseListModel>(
        builder: (context, child, expenses) {
          return ListView.separated(
            itemCount: expenses.items == null ? 1
              : expenses.items.length + 1, itemBuilder: (context,
index) {
              if (index == 0) {
                return ListTile( title: Text("Total expenses:
"
                + expenses.totalExpense.toString(),
                style: TextStyle(fontSize: 24,fontWeight:
FontWeight.bold),) );
              } else {
                index = index - 1; return Dismissible(
                  key:
Key(expenses.items[index].id.toString()),
                  onDismissed: (direction) {
                    expenses.delete(expenses.items[index]);
                    Scaffold.of(context).showSnackBar(
                      SnackBar(
                        content: Text(
```

```

                                "Item with id, " +
expenses.items[index].id.toString()
                                + " is dismissed"
                                )
                            )
                        );
                    },
                    child: ListTile( onTap: () {
                        Navigator.push( context,
MaterialPageRoute(
                            builder: (context) => FormPage(
                                id: expenses.items[index].id,
expenses: expenses,
                            )));
                    },
                    leading: Icon(Icons.monetization_on),
                    trailing: Icon(Icons.keyboard_arrow_right),
                    title: Text(expenses.items[index].category
+ ": " +
\nspent on " +
                    expenses.items[index].amount.toString() + "
                    expenses.items[index].formattedDate,
                    style: TextStyle(fontSize: 18, fontStyle:
FontStyle.italic),))
                );
            },
            separatorBuilder: (context, index) {
                return Divider();
            },
        ),
    ),
    floatingActionButton:
ScopedModelDescendant<ExpenseListModel>(
        builder: (context, child, expenses) {
            return FloatingActionButton(
                onPressed: () {
                    Navigator.push(
                        context, MaterialPageRoute(
                            builder: (context)
=> ScopedModelDescendant<ExpenseListModel>(
                                builder: (context, child, expenses) {
                                    return FormPage( id: 0, expenses:
expenses, );
                                }
                            )
                        );
                    // expenses.add(
                    new Expense(
                        // 2, 1000, DateTime.parse('2019-04-01
11:00:00'), 'Food'
                    )
                );
                // print(expenses.items.length);
            },
            tooltip: 'Increment', child: Icon(Icons.add),
        );

```

```

    }
  )
);
}
}
class FormPage extends StatefulWidget {
  FormPage({Key key, this.id, this.expenses}) : super(key: key);
  final int id;
  final ExpenseListModel expenses;

  @override
  _FormPageState createState() => _FormPageState(id: id, expenses:
expenses);
}
class _FormPageState extends State<FormPage> {
  _FormPageState({Key key, this.id, this.expenses});
  final int id;
  final ExpenseListModel expenses;
  final scaffoldKey = GlobalKey<ScaffoldState>();
  final formKey = GlobalKey<FormState>();
  double _amount; DateTime _date;
  String _category;
  void _submit() {
    final form = formKey.currentState;
    if (form.validate()) {
      form.save();
      if (this.id == 0) expenses.add(Expense(0, _amount, _date,
_category));
      else expenses.update(Expense(this.id, _amount, _date,
_category));
      Navigator.pop(context);
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      key: scaffoldKey, appBar: AppBar(
        title: Text('Enter expense details'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: formKey, child: Column(
            children: [
              TextFormField(
                style: TextStyle(fontSize: 22),
                decoration: const InputDecoration(
                  icon: const Icon(Icons.monetization_on),
                  labelText: 'Amount',
                  labelStyle: TextStyle(fontSize: 18)
                ),
                validator: (val) {
                  Pattern pattern = r'^[1-9]\d*(\.\d+)?$';
                  RegExp regex = new RegExp(pattern);
                  if (!regex.hasMatch(val)) return 'Enter a
valid number';

                  else return null;
                },
                initialValue: id == 0 ? ''
                  : expenses.byId(id).amount.toString(),
                onSave: (val) => _amount = double.parse(val),

```

```

    ),
    TextFormField(
      style: TextStyle(fontSize: 22),
      decoration: const InputDecoration(
        icon: const Icon(Icons.calendar_today),
        hintText: 'Enter date',
        labelText: 'Date',
        labelStyle: TextStyle(fontSize: 18),
      ),
      validator: (val) {
        Pattern pattern = r'^((?:19|20)\d\d)[- /.]
(0[1-9]|1[012])[- /.](0[1-9]|[12][0-
9]|3[01])$';

        RegExp regex = new RegExp(pattern);
        if (!regex.hasMatch(val)) return 'Enter a
valid date';


        else return null;
      },
      onSave: (val) => _date = DateTime.parse(val),
      initialValue: id == 0 ? '' :
expenses.byId(id).formattedDate,
      keyboardType: TextInputType.datetime,
    ),
    TextFormField(
      style: TextStyle(fontSize: 22),
      decoration: const InputDecoration(
        icon: const Icon(Icons.category),
        labelText: 'Category',
        labelStyle: TextStyle(fontSize: 18)
      ),
      onSave: (val) => _category = val,
      initialValue: id == 0 ? '' :
expenses.byId(id).category.toString(),
    ),
    RaisedButton(
      onPressed: _submit,
      child: new Text('Submit'),
    ),
  ],
),
),
),
);
}
}


```

- Now, run the application.
- Add new expenses using floating button.
- Edit existing expenses by tapping the expense entry.
- Delete the existing expenses by swiping the expense entry in either direction.


Some of the screen shots of the application are as follows –




 Enter expense details

 Amount

200

 Date

2019-04-01

 Category

Travel

Submit

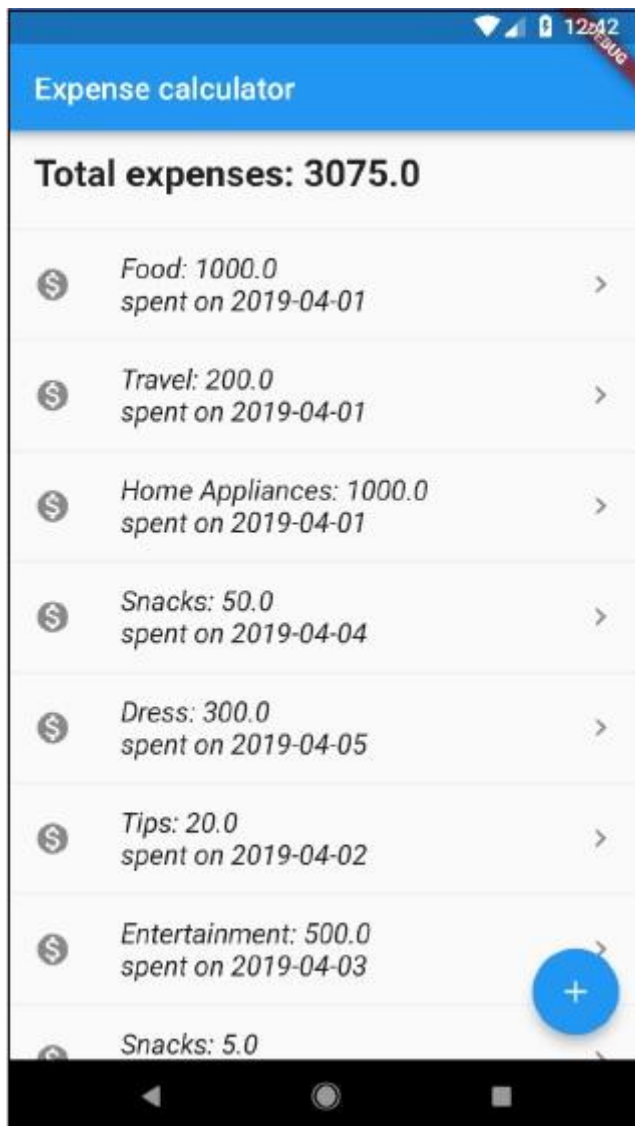
Travel | Traveling | Travels

q w e r t y u i o p

a s d f g h j k l

⬅ z x c v b n m ➡

?123 , . EN • ES ✓



Flutter - Conclusion

Flutter framework does a great job by providing an excellent framework to build mobile applications in a truly platform independent way. By providing simplicity in the development process, high performance in the resulting mobile application, rich and relevant user interface for both Android and iOS platform, Flutter framework will surely enable a lot of new developers to develop high performance and feature-full mobile application in the near future.