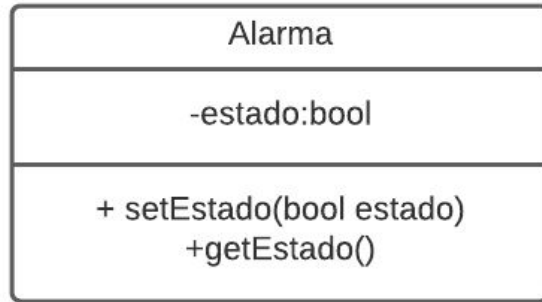


# Estado de Objetos

```
class Alarma {  
    bool _estado;  
  
    Alarma(this._estado);  
}
```

```
void main() {  
    var alarma = Alarma(true);  
  
    print("Estado 1 : $alarma._estado");  
  
    alarma._estado = false;  
  
    print("Estado 2 : $alarma._estado");  
}
```



```
var alarma = Alarma(true);
```

**Evento**



```
alarma. estado = false;
```

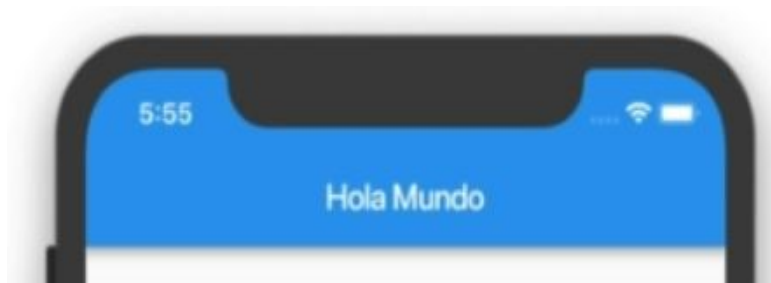
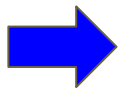
# ¿Qué es un Widget?

Es una clase

Es la unidad básica para construir interfaces de usuario UI.

Los Widgets describen cómo debería ser la vista, dada su configuración y estado actuales.

```
AppBar(  
  title: Text('Hola Mundo'),  
)
```



# ¿Qué es un Widget?

En Flutter, (casi) todo es un widget y los widgets son clases de Dart que describen la vista.

**Son esquemas que Flutter usará para pintar elementos en la pantalla.**

Un widget (o componente) es una clase que define un componente específico de una interfaz de usuario. Para construir una aplicación, se construyen muchos widgets (o componentes) y se arman de diferentes maneras para componer gradualmente widgets más grandes.

Un widget puede definir cualquier aspecto de la vista de una aplicación. Algunos widgets como Row definen aspectos del layout. Algunos son menos abstractos y definen aspectos estructurales, como Button y TextField .

# Tipos de Widgets

**\*Son clases Abstractas**

**StatelessWidget**



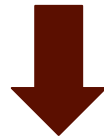
**Sin Estado**

**\*No tiene ningún estado interno que cambie durante la vida del widget.**

**\*Variables de Instancia Finales**

**Ejemplo: Un Botón**

**StatefulWidget**



**Con Estado**

**\*Tiene un estado interno y puede manejar ese estado.**

**Ejemplo: Un Text cuyo valor cambia cuando presionamos un botón**

# Sin Estado y Con Estado

**Un widget es sin estado(StatelessWidget) o con estado(StatefulWidget)**

**Un widget sin estado nunca cambia.**

**Cuando se está ejecutando la aplicación. Un widget es con estado si:**

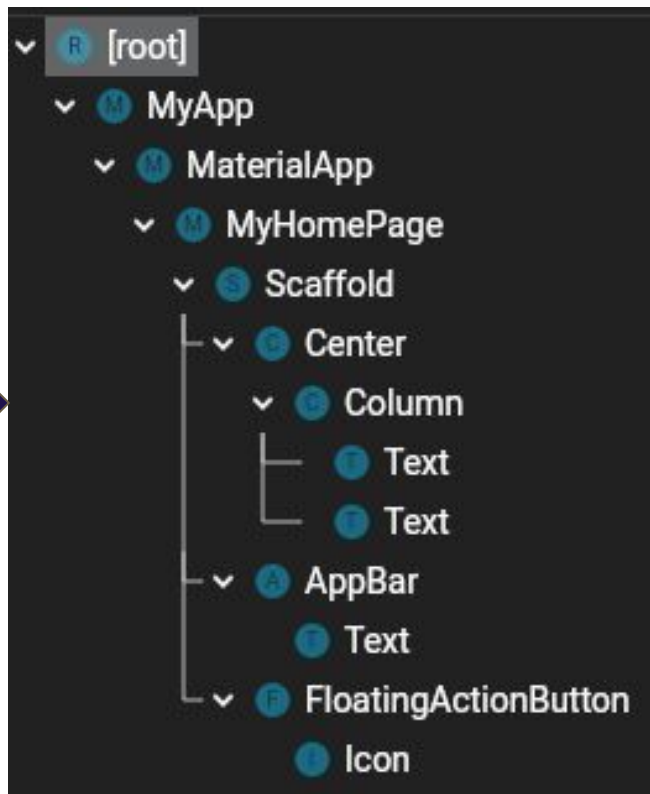
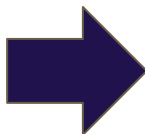
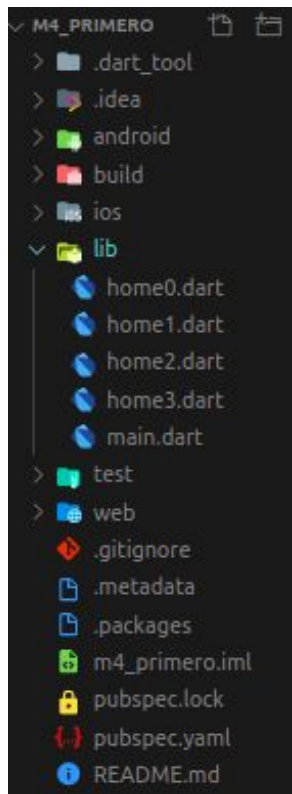
- Puede cambiar su apariencia**

- por eventos que se producen**

- por las interacciones del usuario o cuando recibe datos**

**(los eventos pueden cambiar el estado de los objetos de una clase)**

# Proyecto con Stateful Widgets

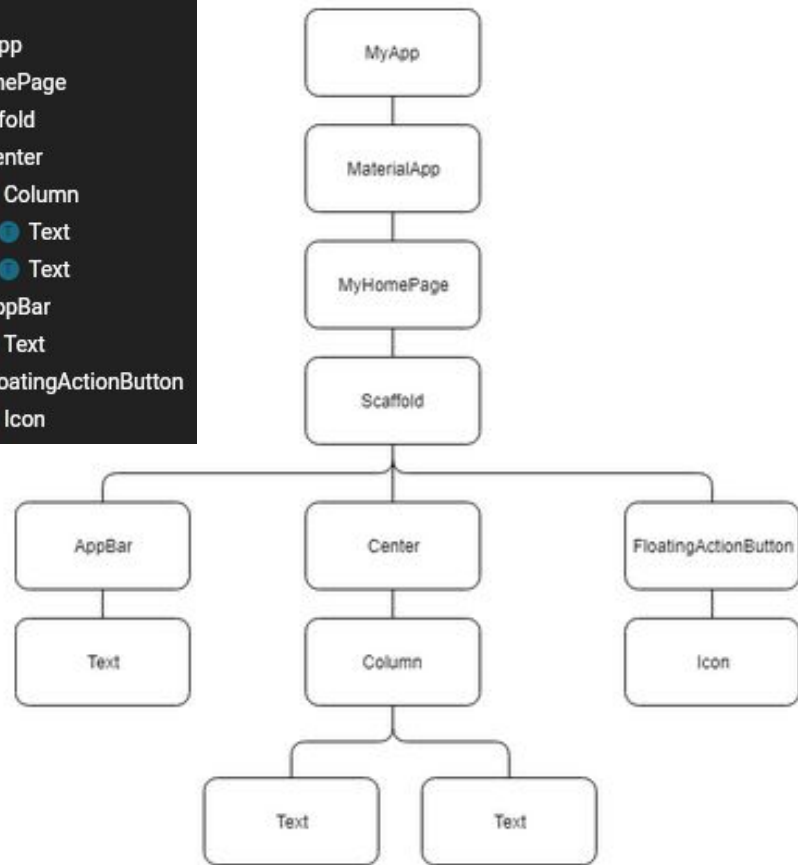


# Árbol de Widgets

Los widgets se organizan en estructura(s) de árbol.

Un widget que contiene otros widgets se denomina Widget padre (o contenedor de Widgets). Los widgets que están contenidos en un Widget padre se llaman Widgets hijos.

**Veamos esto con la aplicación base que se genera automáticamente por Flutter.**

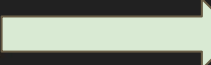


# Proyecto Stateful Widgets

```
import 'package:flutter/material.dart';
import 'package:m4_primer0/home0.dart';

Run | Debug
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Cambiar Estado',
      theme: ThemeData(
        primarySwatch: Colors.orange,
        textTheme: TextTheme(
          headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),
          headline6: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
        ), // TextTheme
      ), // ThemeData
      home: MyHomePage(title: 'Flutter Cambiar de Estado'),
    ); // MaterialApp
  }
}
```



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

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

  final String title;

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

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

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
```



# Proyecto con Stateful Widgets

```
import 'package:flutter/material.dart';
import 'package:m4_primer/home.dart';

Run [Debug]
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Cambiar Estado',
      theme: ThemeData(
        primaryColor: Colors.orange,
        accentColor: Colors.green,
        textTheme: TextTheme(
          bodyText2: TextStyle(color: Colors.purple),
          headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),
          headline6: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
        ),
      ),
      home: MyHomePage(title: 'Flutter Cambiar de Estado'),
    ); // MaterialApp
  }
}
```

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

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

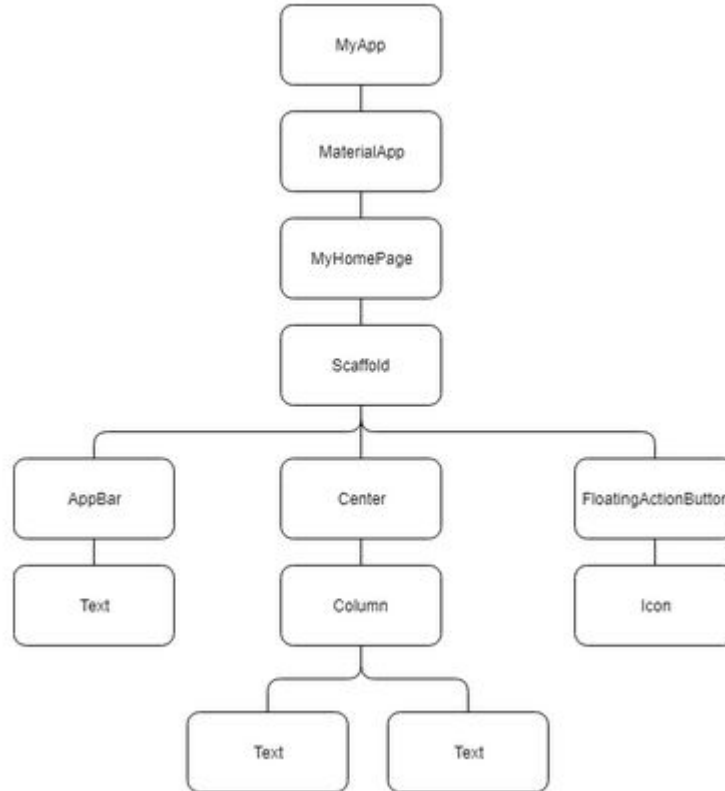
  final String title;

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

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

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widgets> [
            Text(
              'Resultado',
              style: Theme.of(context).textTheme.headline1,
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline1,
            ),
          ], // <Widgets>
        ), // Column
      ), // Center
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Incrementar',
        child: Icon(Icons.add),
      ), // This trailing comma makes auto-formatting nicer for build methods.
    ); // Scaffold
  }
}
```



Flutter Cambiar de Estado

*Resultado*  
0

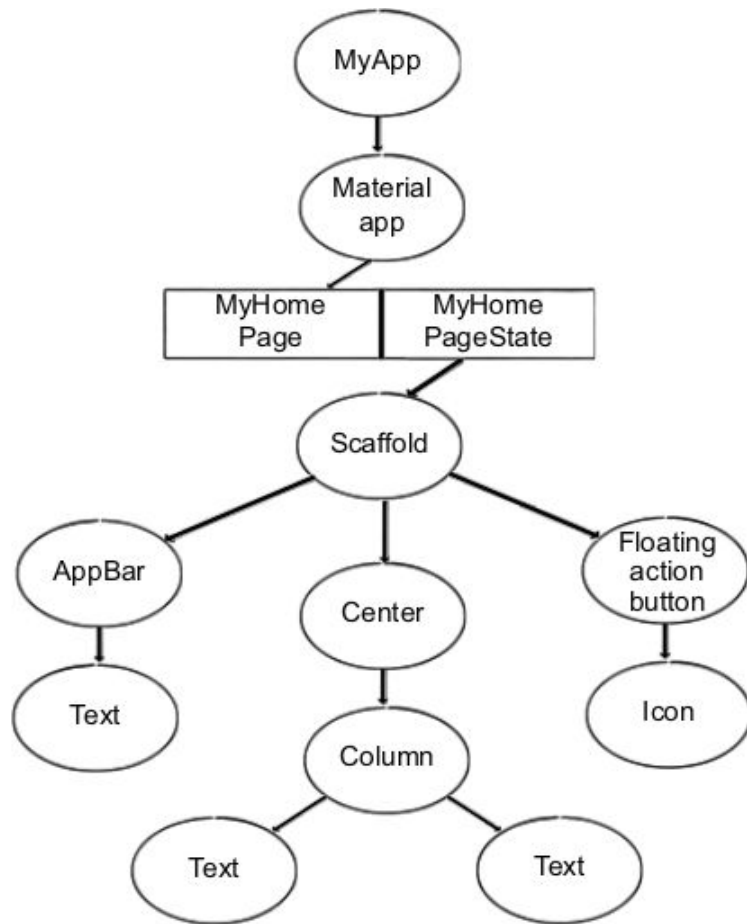


# StatefulWidget

Un **StatefulWidget** tiene un estado interno y puede manejar ese estado.

El nodo del árbol `MyHomePage` está conectado al nodo del árbol `MyHomePageState`.

Las instancias de `StatefulWidget` tienen dos clases involucradas



# StatefulWidget

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

Hereda de StatefulWidget

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

Cada StatefulWidget debe tener un método createState que retorna un State object

```
class _MyHomePageState extends State<MyHomePage> {
```

La clase State hereda del objeto State de Flutter

```
  int _counter = 0;  
  void _incrementCounter() {  
    setState() {  
      _counter++;  
    }  
  }  
}
```

El objeto state llama a **setState()**, indicando al framework que redibuje el widget usando **Build()**.

```
  @override  
  Widget build(BuildContext context) {
```

StatefulWidget requiere un método build

# Noción de Contexto

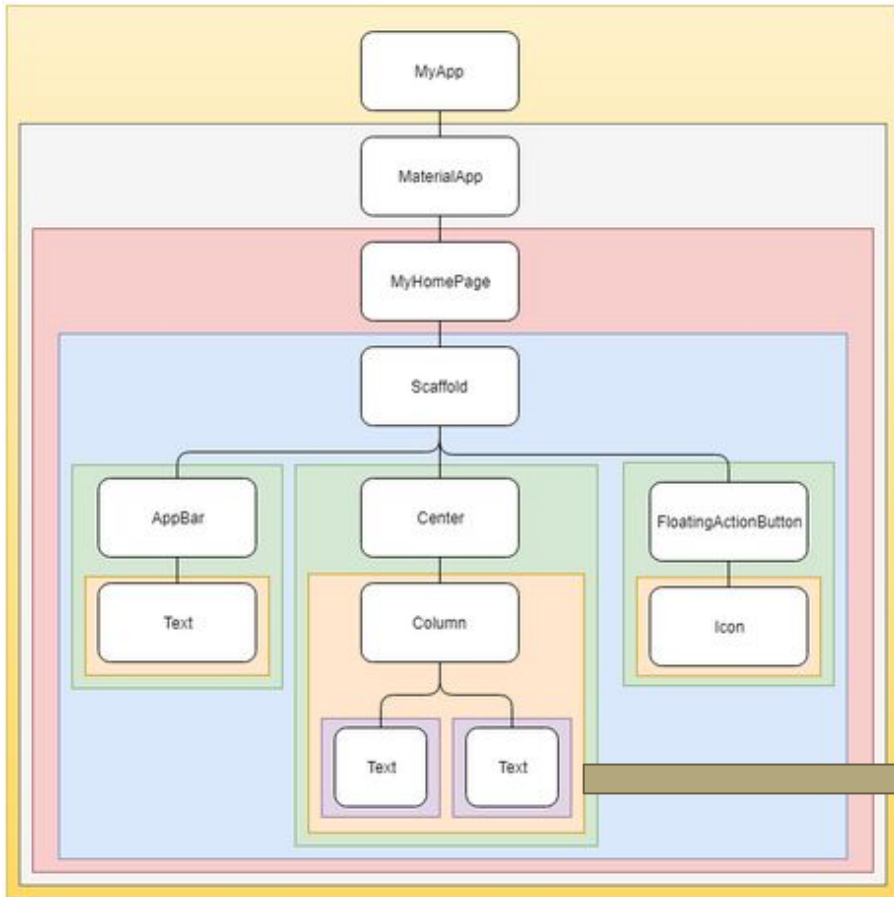
**El contexto es una referencia a la ubicación de un Widget dentro de la estructura de árbol de todos los Widgets que se construyen.**

**Un contexto sólo pertenece a un Widget.**

**Si un widget 'A' tiene widgets hijos, el contexto del widget 'A' se convertirá en el contexto padre de los contextos hijos directos.**

**Los contextos están encadenados y están componiendo un árbol de contextos (relación padre-hijo).**

# Noción de Contexto



```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Cambiar Estado',
    theme: ThemeData(
      primaryColor: Colors.orange,
      accentColor: Colors.green,
      textTheme: TextTheme(
        bodyText2: TextStyle(color: Colors.purple),
        headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),
        headline6: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
      ), // TextTheme
    ), // ThemeData
    home: MyHomePage(title: 'Flutter Cambiar de Estado'),
  ); // MaterialApp
}
```

```
Text(
  '$_counter',
  style: Theme.of(context).textTheme.headline1,
), // Text
// Widgets [1
```

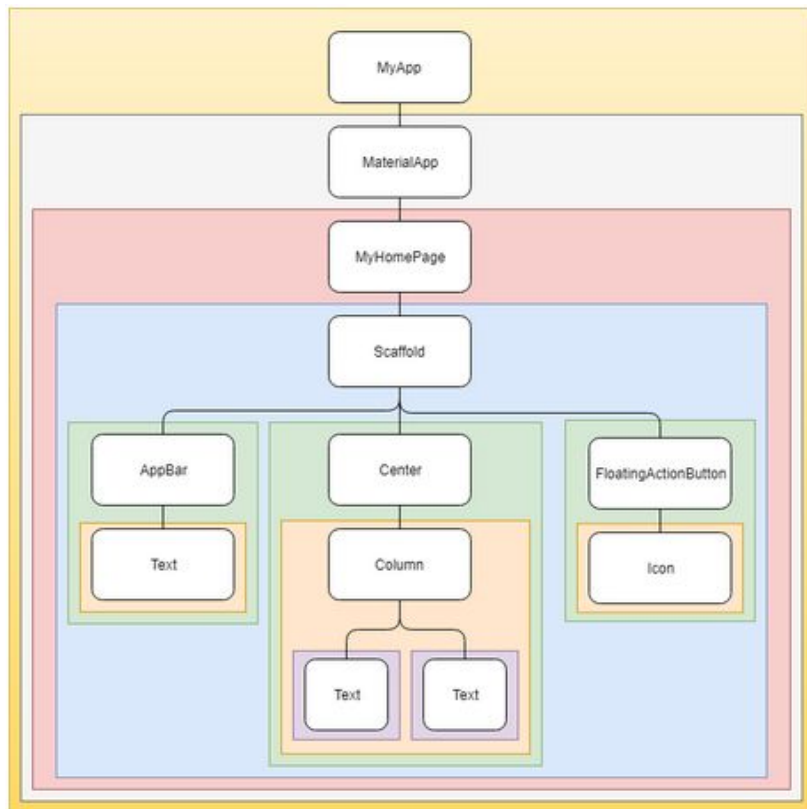
# BuildContext

```
@override  
Widget build(BuildContext context) {
```

Cada método de construcción de un widget (**build**) toma un argumento, **BuildContext**, que es una referencia a la ubicación de un widget en el árbol de widgets.

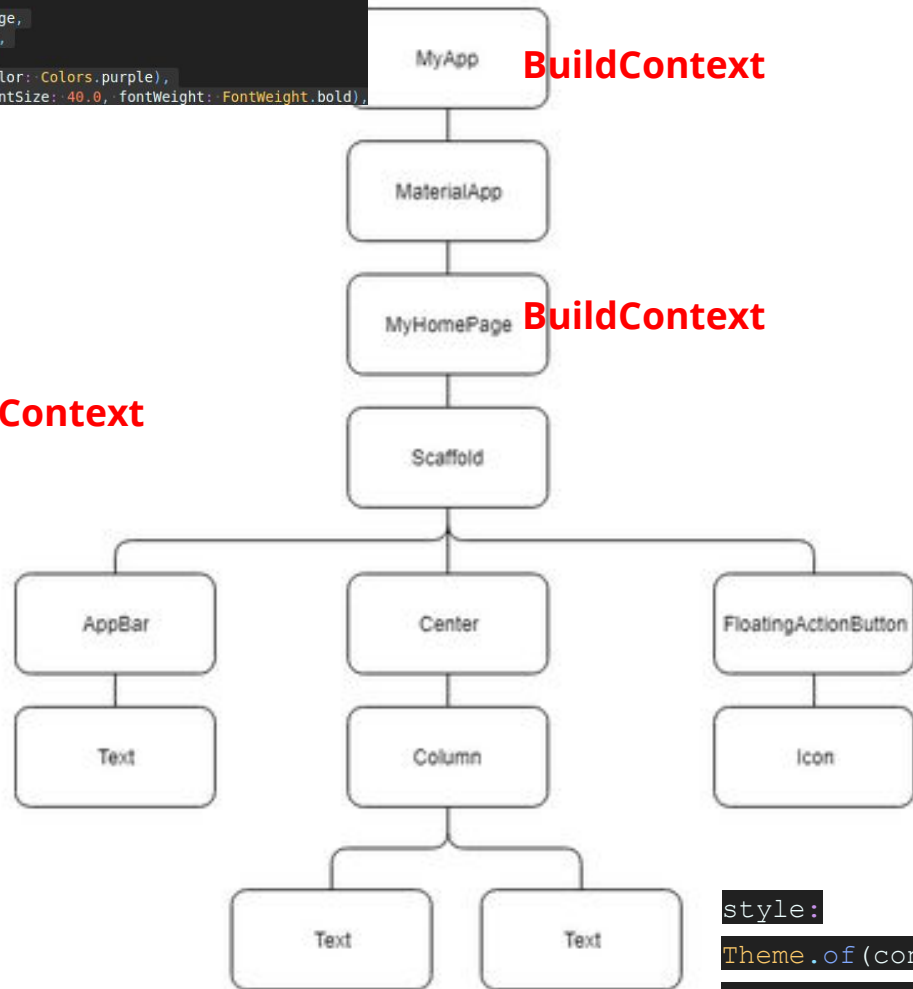
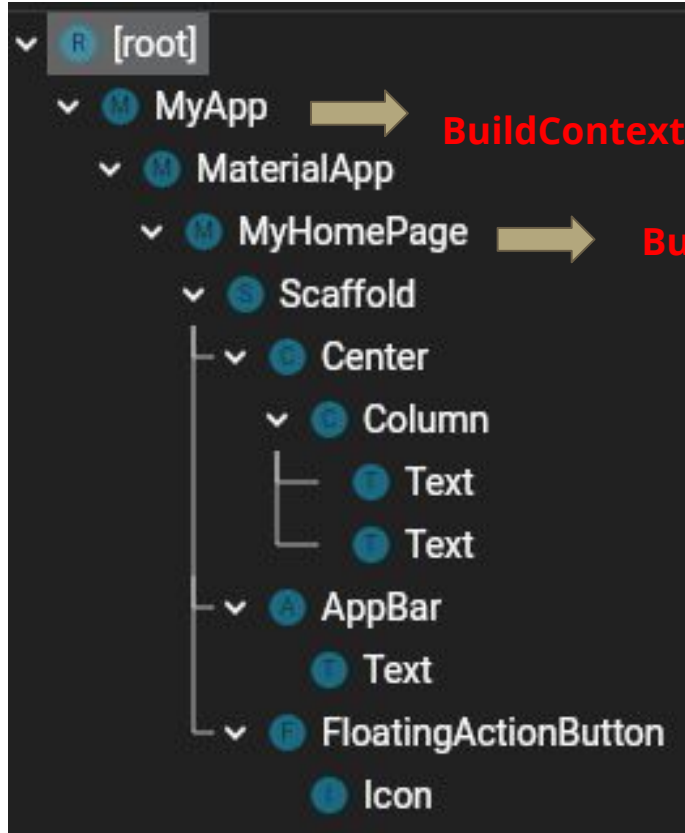
Contiene información sobre el lugar de un widget en el árbol de widgets, no sobre el propio widget.

Cada widget tiene su propio contexto de construcción



# BuildContext

```
theme: ThemeData(  
  primaryColor: Colors.orange,  
  accentColor: Colors.green,  
  textTheme: TextTheme(  
    bodyText2: TextStyle(color: Colors.purple),  
    headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),  
  ),  
)
```



```
style:  
Theme.of(context).textTheme.  
headline1,
```

# ThemeData

Se utiliza para establecer los colores de fondo y estilos de fuente para AppBars, Buttons, Checkboxes, y más.

**Theme.of(context).** Buscará en el árbol de widgets (Widget tree) y devolverá el tema más cercano en el árbol.



**primaryColor: Colors.orange** (El color de fondo de la mayoría de las partes de la aplicación (barras de herramientas, barras de pestañas, etc.))



**bodyText2: TextStyle(color: Colors.purple)**

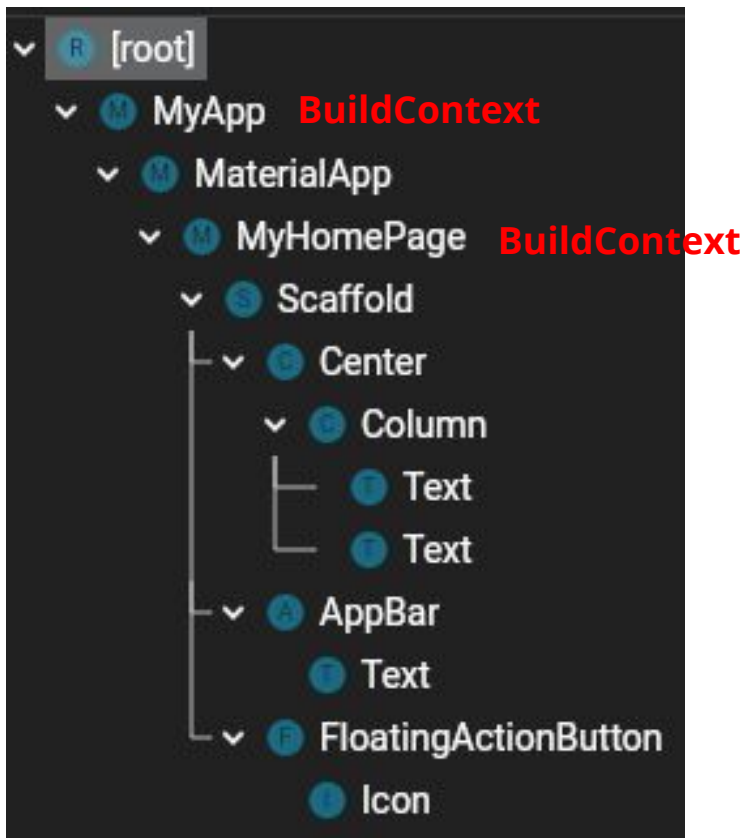


**accentColor: Colors.green**(El color de primer plano para los widgets)

Fuente: <https://api.flutter.dev/flutter/material/ThemeData-class.html>



# ThemeData



```
theme: ThemeData(  
  primaryColor: Colors.orange,  
  accentColor: Colors.green,  
  textTheme: TextTheme(  
    bodyText2: TextStyle(color: Colors.purple),  
    headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),  
    headline6: TextStyle(fontSi 36.0, fontStyle: FontStyle.italic),  
  ), // TextTheme
```



```
Text(  
  '$_counter',  
  style: Theme.of(context).textTheme.headline1,  
), // Text
```

# Usando un Tema (Theme)

Usar un tema dentro de los métodos build() de los widgets usando el método Theme.of(context).

**El método Theme.of(context)** busca en el árbol de los widgets y devuelve el Tema más cercano en el árbol. Si tienes un Tema independiente definido sobre tu widget, se devuelve. Si no, se devuelve el tema de la aplicación.

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Cambiar Estado',
    theme: ThemeData(
      primaryColor: Colors.orange,
      accentColor: Colors.green,
      textTheme: TextTheme(
        bodyText2: TextStyle(color: Colors.purple),
        headline1: TextStyle(fontSize: 40.0, fontWeight: FontWeight.bold),
        headline6: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
      ), // TextTheme
    ), // ThemeData
    home: MyHomePage(title: 'Flutter Cambiar de Estado'),
  ); // MaterialApp
}
```



```
Text(
  '$_counter',
  style: Theme.of(context).textTheme.headline1,
), // Text
```

# State<T extends StatefulWidget> class

```
class MyHomePage extends StatefulWidget {  
  MyHomePage({Key key, this.title}) : super(key: key);  
  
  final String title;  
  
  @override  
  _MyHomePageState createState() => _MyHomePageState();  
}  
  
class _MyHomePageState extends State<MyHomePage> {
```

El estado es una información que (1) puede ser leída sincrónicamente cuando se construye el widget y (2) puede cambiar durante la vida del widget.

Es responsabilidad del programador del widget asegurarse de que el Estado sea notificado rápidamente cuando dicho estado cambie usando **State.setState**.

```
setState(() {  
  _counter++;  
});
```

# Cambio de Estado

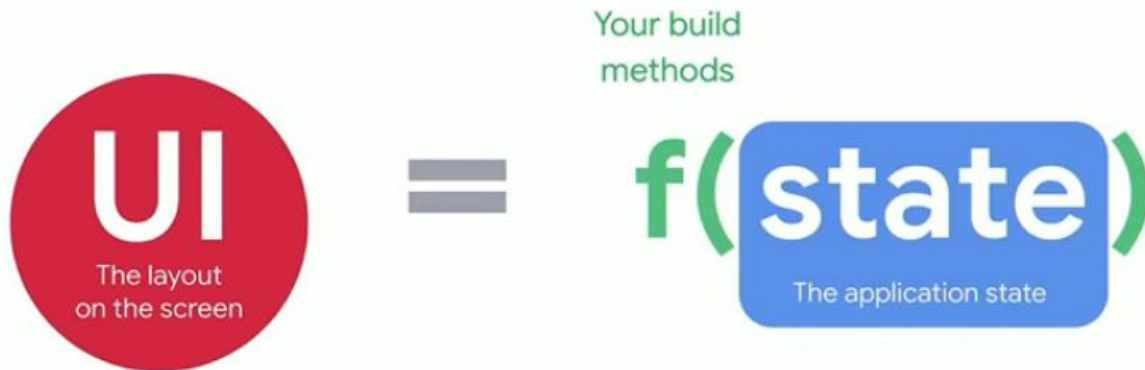


# Estado de un Widget

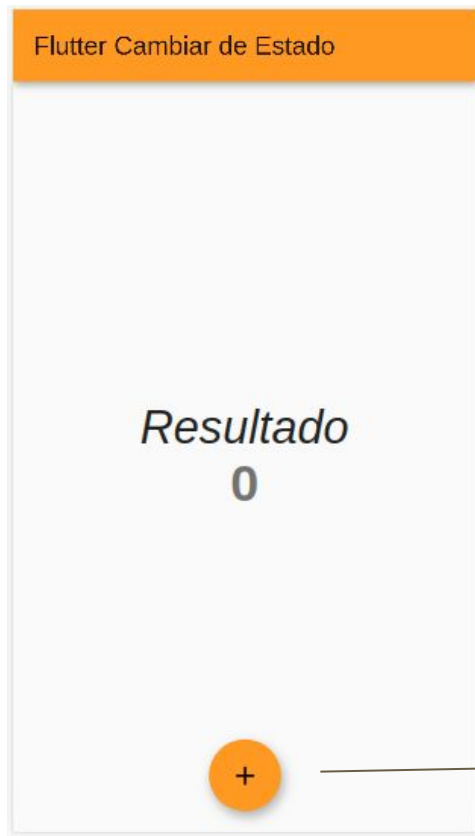
El estado de un widget se almacena en un objeto State, separando el estado del widget de su apariencia.

Si es necesario que el estado del Widget cambie, tenemos que producir que el objeto state llame a **setState()**, indicando al framework que redibuje el widget usando **Build()**.

El estado consiste en valores que pueden cambiar, como el valor de un campo de texto o los ítems de una lista.



# State



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

```
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }
```

```
  @override  
  Widget build(BuildContext context) {  
    print("build Repinta App");  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
      ),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            Text(  
              'Resultado',  
              style: Theme.of(context).textTheme.headline6,  
            ),  
            Text(  
              '$_counter',  
              style: Theme.of(context).textTheme.headline1,  
            ),  
          ],  
        ),  
      ),  
      floatingActionButtonLocation: FloatingActionButtonLocation.centerFloat,  
      floatingActionButton: FloatingActionButton(  
        onPressed: _incrementCounter,  
        tooltip: 'Incrementar',  
        child: Icon(Icons.add),  
      ), // This trailing comma makes auto-formatting nicer for build methods.  
    );  
  }
```

2

3

1

# StatelessWidget vs StatefulWidget

**StatelessWidget** es un Widget que se construye solo con configuraciones que no cambian en tiempo de ejecución.

Por ejemplo, creamos una aplicación que contiene Text con el número 10. Entonces nuestra aplicación no tiene una función para cambiar ese número. Entonces, lo que se usa aquí es StatelessWidget.

**StatefulWidget** es un widget que puede cambiar dinámicamente.

La función `setState ()` le dirá al framework que un objeto ha cambiado el Estado, luego re-construirá el Widget.

# Resumen

Si queremos que la pantalla cambie dinámicamente mientras la aplicación se está ejecutando, utilizaremos el `StatefulWidget` .

En otro caso, usar `StatelessWidget`.

**\*=>Luego vemos que podemos usar un framework como GetX para evitar el uso de `StatefulWidget`**



# Gestión del Estado (State management)

Flutter es declarativo. Esto significa que Flutter construye su interfaz de usuario para reflejar el estado actual de la aplicación.

Cuando el estado de la aplicación cambia (por ejemplo, el usuario presiona un botón en la pantalla), cambia el estado, y eso desencadena un redibujo de la interfaz de usuario. No hay un cambio imperativo de la propia interfaz (como en el caso de `widget.setText`). Se cambia el estado, y la interfaz se reconstruye desde cero.

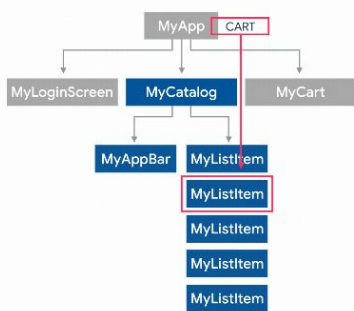
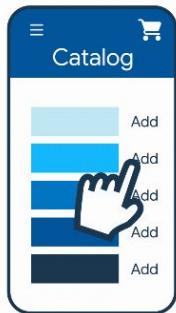
$$\text{UI} = f(\text{state})$$

The layout  
on the screen

Your  
build  
methods

The application state

# Gestión del Estado (State management)



## Estado Efímero

El estado efímero (a veces llamado estado de UI o estado local) es el estado que se puede contener ordenadamente en un solo widget.

A medida que trabajamos con Flutter, llega un momento en que necesitamos compartir el estado de la aplicación entre las pantallas, a través de la aplicación.

**Hay muchos enfoques que se pueden utilizar.**

## Estado de la App

El estado no es efímero, se necesita compartir el estado a través de muchas partes de la aplicación (estado compartido). Ej: Carrito de compras

# Propuestas para la Gestión del Estado

La gestión del Estado es un tema complejo.

El enfoque que se va utilizar depende del casos de uso.

## Propuestas:

**setState** -> Lo que necesitamos cambiar de estado tiene que estar dentro de una clase de tipo state

**Provider** (<https://pub.dev/packages/provider>)

**Bloc** ([https://pub.dev/packages/flutter\\_bloc](https://pub.dev/packages/flutter_bloc))

 **GetX** (<https://pub.dev/packages/get>)

# Propuestas para la Gestión del Estado (GetX)

## setState

Lo que necesitamos cambiar de estado tiene que estar dentro de una clase de tipo state. Cada vez que se llama setState se llama a Build que redibuja todo lo que tiene contenido. Pérdida de performance con Widgets que consumen recursos.

## Provider

Permite divide la lógica del negocio de la vista. Menos código que en BLoC. problema: Hay Widgets que están escuchando el estado del Provider. Cuando se produce un cambio en el provider los Widgets que están escuchando se redibujan enteros. No permite decidir que partes específicas del Widget es necesario redibujar.

## BLoC (Patrón)

Permite divide la lógica del negocio de la vista. Buen rendimiento de la aplicación porque nos permite indicar que partes de la vista se pueden redibujar. Permite estados globales. Buen manejo de eventos como en Redux. Problemas: Demasiado código y curva de aprendizaje mayor a GetX. Difícil de comprender para principiantes. Similar a Redux. BLoC con Cubit??



Es una solución extra-liviana y poderosa para Flutter. Microframework.

Combina la **gestión de estado de alto rendimiento** {reactivo}, la **inyección de dependencia inteligente** y la **gestión de rutas de forma rápida y práctica**.

**PERFORMANCE:** GetX se centra en el rendimiento y el consumo mínimo de recursos.

**PRODUCTIVIDAD:** GetX utiliza una sintaxis fácil y agradable.

**ORGANIZACIÓN:** GetX permite el desacoplamiento total de la vista de la lógica del negocio.

# GetX

## Instalación

**dependencies:**

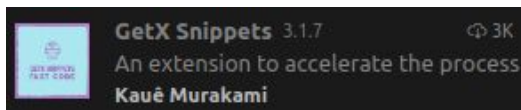
**get:**

```
{...} pubspec.yaml
19
20   environment:
21     sdk: ">=2.7.0 <3.0.0"
22
23   dependencies:
24     flutter:
25       sdk: flutter
26     get:
```

**Usar con:**

**import 'package:get/get.dart';**

## Instalar Extensión para Visual Studio Code



# Gestión de Estado Reactiva

**\*GetX convierte la programación reactiva en algo fácil de aprender**

**Ejemplo. Esta es tu variable "nombre":**

```
var nombre = 'Prueba1';
```

**Para que sea observable, solo necesita agregar ".obs" al final:**

```
var nombre = 'Prueba1'.obs;
```

**¿StreamBuilder? ¿initialValue? ¿builder? No, solo necesitas jugar con esta variable dentro de un widget Obx.**

```
Obx(() => Text("$nombre"))
```

**\*Internamente - ¿Cómo funciona? Se crea un stream de Strings, asignamos el valor inicial a nombre de "Prueba1", advertimos a todos los widgets que usan el observable nombre que ahora cuando se modifique nombre, ellos también lo harán.**

# Cambio de Estado

```
class Home extends StatelessWidget {  
  var nombre = 'Prueba1'.obs;  
}
```



Click a un Botón

```
floatingActionButton: FloatingActionButton(  
  child: Icon(Icons.add_comment),  
  onPressed: () => nombre.value = 'Prueba2',  
)); // FloatingActionButton // Scaffold
```

```
Obx(() => Text("$nombre")),
```

```
@override  
Widget build(context) => Scaffold(  
  appBar: AppBar(title: Text("App Reactiva")),  
  body: Center(  
    child: Obx(() => Text("$nombre")),
```



# Ejercicio con GetX

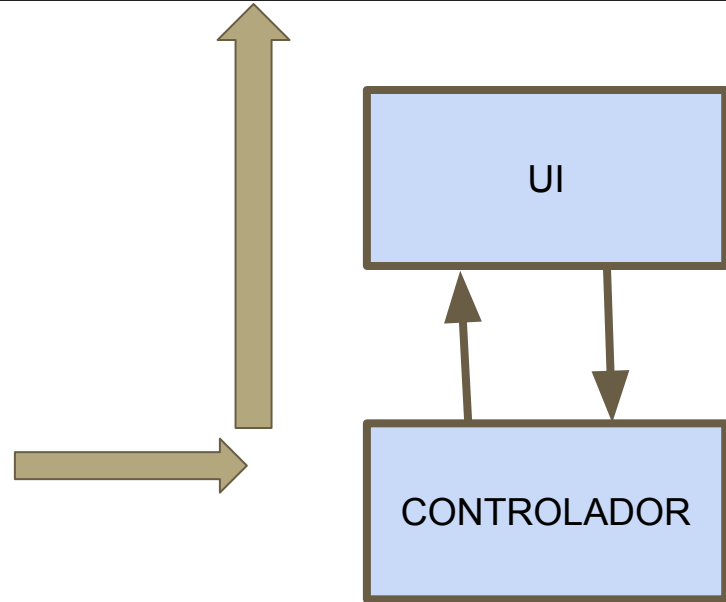
```
void main() => runApp(GetMaterialApp(home: Home()));
```

```
class Controller extends GetxController {  
  var count = 0.obs;  
  increment() => count.value++;  
}
```

```
class Home extends StatelessWidget {  
  @override  
  Widget build(context) {  
    // Cree una instancia de su clase usando Get.put() para que esté disponible  
    final Controller c = Get.put(Controller());  
  
    return Scaffold(  
      // Utilice Obx(()=> para actualizar Text() siempre que se cambie el rec  
      appBar: AppBar(title: Obx(() => Text("Clicks: " + c.count.string))),  
      // Reemplace el Navigator.push de 8 líneas por un simple Get.to(). No n  
      body: Center(  
        child: RaisedButton(  
          child: Text("Go to Other"), onPressed: () => Get.to(Other()))),  
      floatingActionButton: FloatingActionButton(  
        child: Icon(Icons.add), onPressed: c.increment)); // FloatingAction  
    }  
}
```

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

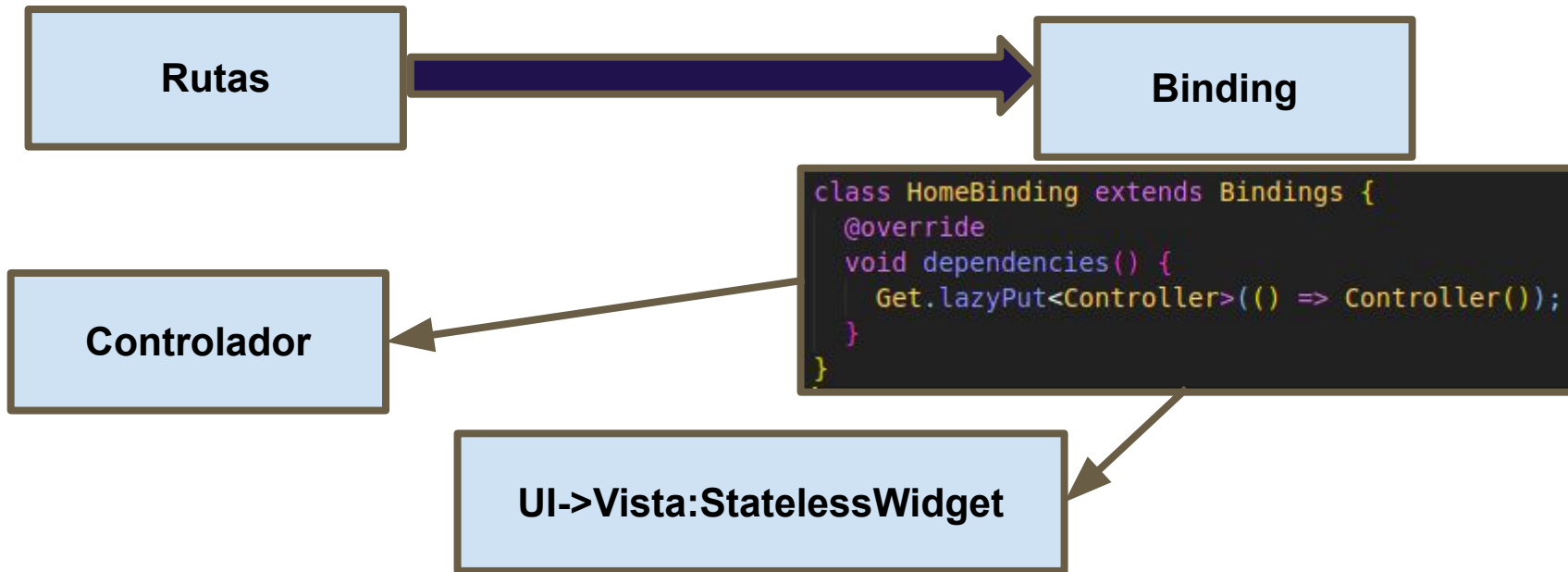
```
class Other extends StatelessWidget {  
  // Puede pedirle a Get que busque un controlador que está si  
  final Controller c = Get.find();  
  
  @override  
  Widget build(context) {  
    // Acceder a la variable de recuento actualizada  
    return Scaffold(body: Center(child: Text(c.count.string)));  
  }  
}
```



# Binding (Vinculador)

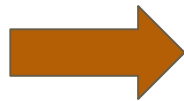
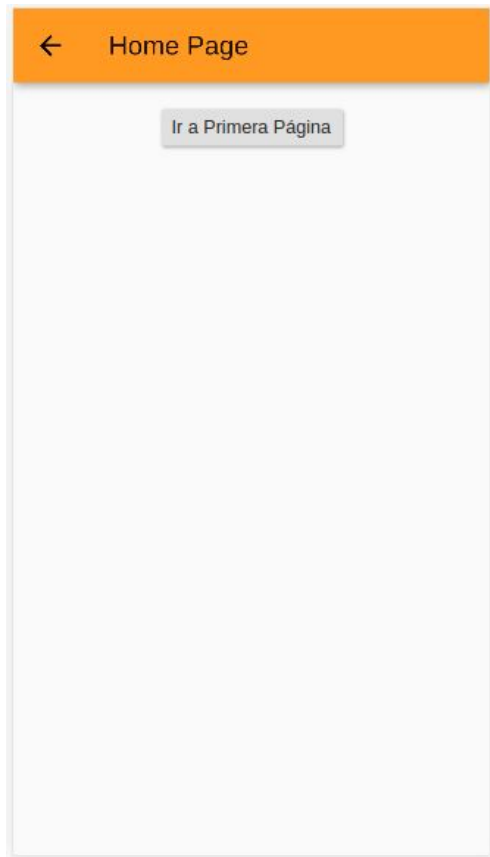
La clase Binding permite desacoplar la inyección de dependencia al tiempo que vincula rutas con el gestor de estado y el gestor de dependencias. Elimina:

```
final Controller c = Get.put(Controller());
```



**Elimina la Necesidad de usar:** `final Controller c = Get.put(Controller());`

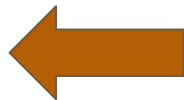
# Rutas y Navegación



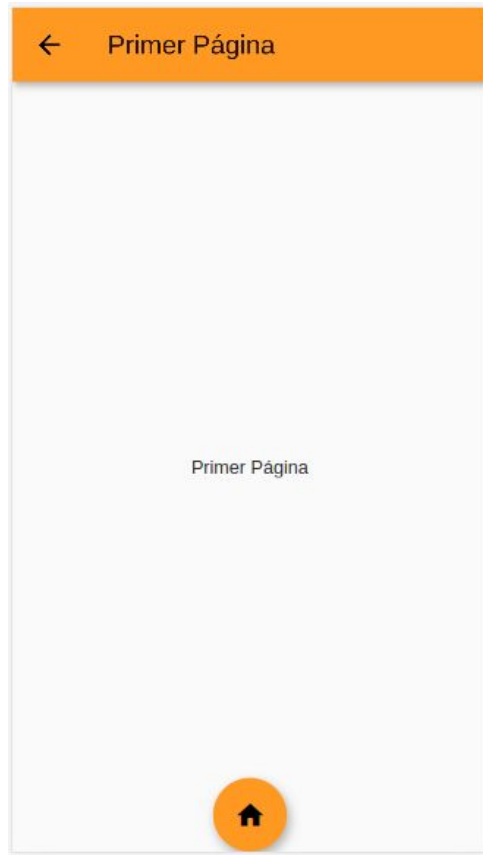
**Get.to(PrimerPagina());**

// Navegador de Flutter

```
Navigator.push(  
  MaterialPageRoute(  
    builder: (_) {  
      return PrimerPagina();  
    },  
  ),  
);
```

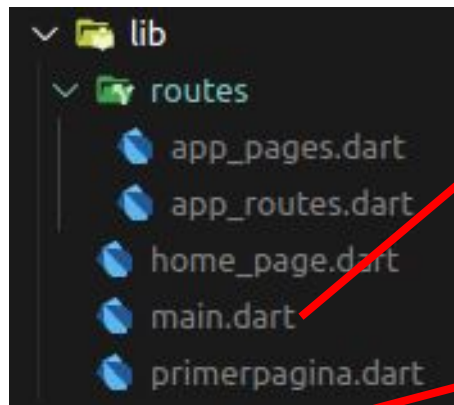


**Get.to(HomePage());**



# Rutas

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



```
void main() {  
  runApp(GetMaterialApp(  
    debugShowCheckedModeBanner: false,  
    initialRoute: '/',  
    theme: ThemeData(  
      primarySwatch: Colors.orange,  
      visualDensity: VisualDensity.adaptivePlatformDensity,  
    ), // ThemeData  
    defaultTransition: Transition.rightToLeftWithFade,  
    getPages: AppPages.pages,  
    home: HomePage(),  
    //initialBinding: SplashBinding(),  
  )); // GetMaterialApp  
}
```

```
class AppPages {  
  static final List<GetPage> pages = [  
    GetPage(  
      name: AppRoutes.HOMEPAGE,  
      page: () => HomePage(),  
      //binding: HomeBinding(),  
    ), // GetPage  
    GetPage(  
      name: AppRoutes.PRIMERPAGINA,  
      page: () => PrimerPagina(),  
      //binding: DetailBinding(),  
    ) // GetPage  
  ];  
}
```

```
class AppRoutes {  
  static const HOMEPAGE = "home";  
  static const PRIMERPAGINA = "primerpagina";  
}
```

```
Get.toNamed(AppRoutes.PRIMERPAGINA);
```

# Navegación con rutas nombradas

Para navegar a la siguiente pantalla

```
Get.toNamed("/NextScreen");
```

Para navegar y eliminar la pantalla anterior del árbol

```
Get.offNamed("/NextScreen");
```

Para navegar y eliminar todas las pantallas anteriores del árbol

```
Get.offAllNamed("/NextScreen");
```

# Enviar datos a rutas nombradas

Envía lo que quieras usando el parámetro `arguments`. `GetX` acepta cualquier cosa aquí, ya sea un `String`, `Map`, `List` o incluso una instancia de clase.

`Get.toNamed("/NextScreen", arguments: 'Get is the best');`

luego en su clase o controlador:

`print(Get.arguments);`

```
Get.toNamed(AppRoutes.PRIMERPAGINA,  
  arguments: 'Parametro de Home Page');
```



```
child: (Get.arguments != null)  
  ? Text(Get.arguments)  
  : Text("No llega parámetro")),
```

# Ejercicio

**En HomePage() crear un tercer botón para navegar a una cuarta página que hay que crear**