# 1. Bienvenida e Introducción

## ¿Qué es Flutter?

Flutter es un SDK creado por Google, pensado para desarrollar aplicaciones nativas (cross platform) para iOS y Android.

El principal objetivo de Flutter es el diseño de interfaces móviles; sin embargo, esta basado en un lenguaje de programación que permite compartir código fuente para ambas plataformas, lo que facilita el aprovechamiento de la lógica del negocio como un elemento común.

Durante el año 2017 Google hizo la presentación y lanzamiento de Flutter en su versión alpha, luego a principios de 2018 lanzó la primera versión beta y ya para este momento, en diciembre de 2018, está disponible la versión oficial de Flutter 1.0 estable.

Flutter utiliza Dart, un lenguaje de programación también creado por Google y que estaría orientado no solo al desarrollo de aplicaciones móviles sino también de aplicaciones web, incluso se espera que Flutter sea el framework principal para el desarrollo de aplicaciones para Fuschia, el nuevo OS en el que Google se encuentra trabajando.

A diferencia del desarrollo híbrido de aplicaciones móviles basado en un WebView para mostrar código HTML, o el desarrollo bridge que utiliza renderers intermedios de código nativo para la generación de apps, Flutter compila directamente hacia el código final que interactúa con el procesador, saltándose los pasos intermedios, por lo que las aplicaciones resultantes tienen una mejor performance y mayor rendimiento. Flutter usa sus propios Widgets y engine de renderizado llamado Skia Canvas desarrollado en C++.

## Dart y Flutter

Dart es el lenguaje de programación que usa Flutter para operar. Es un lenguaje de programación que usa el paradigma de Programación Orientada a Objetos (OOP). Ha sido creado por Google y es considerado por algunos como “el lenguaje de programación del futuro” por su versatilidad de uso.

Dart se puede usar en dispositivos móviles con Flutter; en el desarrollo web, a través de un framework para el desarrollo frontend con JavaScript llamado AngularDart, que es una combinación con Angular; e incluso se puede usar en desarrollo backend a través de frameworks para HTTP servers como Aqueduct.

Aún cuando no se necesitan conocimientos previos específicos para completar este curso, haber trabajado previamente con Dart, con lenguajes de programación orientada a objetos como Java, C++, etc. o haber desarrollado aplicaciones móviles usando los SDKs de Android y iOS en código nativo, con React Native, Xamarin o Ionic, te hará más fácil el camino.

La sintaxis de Dart se parece bastante a Java. Sin embargo, Google ha declarado que Dart está inspirado totalmente en la programación reactiva y estilo de programación de React.js, por lo que si has trabajado anteriormente con este framework (usando styled components) seguro te será fácil comprender y adoptar la sintaxis de Dart.

## Flutter para desarrolladores Android, iOS y Xamarin.forms

### Android

Todo lo que tenemos en las interfaces de Android como: botones, textos, etiquetas, imágenes, etc., proviene (o es una instancia) de la clase padre View y todos estos elementos en su conjunto, al final, constituirán a su vez un View. El lenguaje de esquematización usado en Android para diseñar y definir la composición de las interfaces es XML.

### iOS

De manera similar, en iOS todo lo que tenemos en las interfaces como: botones, textos, etiquetas, etc., proviene (o es una instancia) de la clase principal UIView. Para la composición y diseño de las interfaces iOS cuenta con un entorno gráfico dedicado a esta tarea cuyo componente principal son los Storyboards.

### Xamarin / Xamarin.Forms

En el ecosistema de desarrollo multiplataforma de Microsoft y .NET, Xamarin / Xamarin.Forms, los elementos que conforman las interfaces de las aplicaciones: botones, textos, imágenes, etc., son instancias de la clase base Element mientras que para el diseño y composición de la interfaz, el lenguaje de esquematización utilizado es XAML (derivado de XML).

### Flutter

En Flutter, el equivalente a los Views, UIViews y Elements, con los que se componen las interfaces de las aplicaciones, son los Widgets, de hecho todo en Flutter es en esencia, un Widget, y la forma en que se diseñan y se componen las interfaces es mediante la definición de árboles de Widgets o Widget tree en los que todos los Widget de la interfaz están organizados de forma anidada y representan atributos o propiedades de este árbol.

## Flutter para desarrolladores React Native

Si desarrollas aplicaciones con React Native ya habrás notado que hacerlo con Flutter es muy similar pues utiliza el mismo estilo reactivo.

La principal diferencia es que mientras que React Native transpila (traduce) el código a Widgets Nativos para cada plataforma, Flutter compila todo directamente a Nativo controlando cada pixel de la pantalla para evitar problemas de rendimiento causados por el bridge de JavaScript.

Ahora que estás interesado en aprender Flutter voy a darte un breve recorrido entre lo que maneja React Native y lo que encontrarás en Flutter.

El punto de entrada de una aplicación en React Native está definido por ti a partir de una función

**// JavaScript**

function startHere() {

// Can be used as entry point

}

Flutter maneja un punto de entrada ya definido que es la función main

**// Dart**

main() {

}

Más allá de esto, verás que la manera de funcionar de cada uno es muy similar.

A continuación te presento cómo sería un Hola Mundo con React Native vs. Flutter:

**// React Native**

import React from "react";

import { StyleSheet, Text, View } from "react-native";

export default class App extends React.Component {

render() {

return (

<View style={styles.container}>

<Text>Hello world!</Text>

</View>

);

}

}

const styles = StyleSheet.create({

container: {

flex: 1,

backgroundColor: "#fff",

alignItems: "center",

justifyContent: "center"

}

});

**// Flutter**

import 'package:flutter/material.dart';

void main() {

runApp(

Center(

child: Text(

'Hello, world!',

textDirection: TextDirection.ltr,

),

),

);

}

Verás que la verbosidad entre códigos es más reducida en Flutter y esto es gracias al lenguaje de programación Dart y en general la mejora que se está proponiendo con este SDK.

# 2. Creando mi entorno de desarrollo

## Composición de un proyecto en Flutter

Una vez instalado el plugin de Flutter en Android Studio veremos habilitada la opción Start a new Flutter project que será la opción la que usaremos para crear nuestro proyecto.

Seguidamente veremos los diferentes tipos de proyectos que podemos crear con Flutter: Flutter Application, Flutter Plugin, Flutter Package o Flutter Module. Para el proyecto que desarrollaremos en este curso escogeremos Flutter Application. Seguidamente indicamos las configuraciones básicas para definir nuestro proyecto como el nombre, ubicación del SDK, ubicación del proyecto, la descripción y el company domain, entre otros.

Una vez completado el proceso de inicialización, Android Studio habrá generado el sistema de archivos del proyecto que contendrá básicamente las siguientes carpetas:

flutter\_app/ : carpeta raíz del proyecto, donde además crearemos entre otras cosas las carpetas donde guardaremos nuestros recursos o assets de la aplicación.

android/ : aquí se almacenarán todos los archivos que corresponden a un proyecto Android. Es en esta ubicación donde se encontrarán los archivos de configuración y demás recursos que pudiéramos querer retocar o modificar en Android Studio para comportamientos particulares de la app en esta plataforma, como el FlutterActivity o el AndroidManifest.

ios/ : similarmente, aquí se almacenarán todos los archivos que corresponden al sistema operativo iOS. Es en esta ubicación donde se encontrarán los archivos de configuración y otros que pudiéramos querer retocar o ajustar en XCode para comportamientos específicos relacionados con esta plataforma, como el FlutterAppDelegate o el info.plist, entre otros.

lib/ : es aquí donde encontramos todos los archivos que corresponden a nuestro proyecto con Dart y donde vive la aplicación Flutter. Todos los archivos que estaremos generando durante el curso se almacenarán en esta carpeta.

Hay un archivo especial llamado pubspec.yaml donde se guardan las configuraciones del proyecto. Este archivo está escrito en el lenguaje YAML, que es un formato de serialización de datos legible por humanos inspirado en XML. En este archivo tendremos que declarar todos los recursos externos o assets que vamos a utilizar en nuestro proyecto como: fuentes de letras, imágenes, etc.

# 3. Interfaces en Flutter

## Programación Declarativa en Flutter

El estilo de programación que utiliza Flutter se llama Programación Declarativa, inspirado en el estilo de programación de React y que va de la mano con el paradigma de Programación Funcional.

Existen dos tipos de estilos de programación:

Programación Imperativa: es el estilo de programación que más se utiliza en lenguajes de programación tradicionales como Lenguaje C, Java, PHP o JavaScript. En general la programación imperativa se fundamenta en el uso de métodos de clases que reciben parámetros o argumentos sencillos para realizar tareas específicas.

Programación Declarativa: este estilo de programación utiliza como parámetros estructuras un poco más complejas formadas por datos compuestos, objetos, pares propiedad-valor, etc. que en cierta forma es más fácil de leer y más comprensible para el desarrollador. Este estilo de programación es el que usa Flutter.

Básicamente ambos estilos de programación hacen lo mismo, producen los mismos resultados, su diferencia radica esencialmente en la legibilidad y la organización del código.

## Estructura de un programa en Flutter

Partiendo de la estructura básica creada por Android Studio para el tipo de proyecto Flutter Application, la primera línea de código que generalmente nos vamos a encontrar en nuestros archivos .dart es:

import 'package:flutter/material.dart';

Con esta instrucción importaremos a nuestro código el paquete de todos los Widgets (o la mayoría de ellos) de Material Design que son la base principal de clases con la que trabaja Flutter.

Similar a como es en Java o C++, en Dart, el método main() será el punto de inicio de nuestra aplicación. Por su parte, runApp() es la función que inicializa y da vida a la aplicación definida por el Widget App. En otras palabras, runApp() toma el widget y lo sirve.

App() es la clase principal que representa la raíz de la aplicación. Esta clase extiende o hereda de StatelessWidget que es la clase padre de todos los widgets. App() posee un método heredado llamado build() encargado de construir o mostrar todo el árbol de widgets anidados (o Widgets tree) que hayamos definido con MaterialApp(), que es la clase (que actúa wrapper) que implementa Flutter para dar vida, forma y apariencia a todos los widgets de nuestra interfaz de usuario.

La propiedad fundamental del widget MaterialApp en la que definimos toda la estructura base de nuestra aplicación es home: a la que a su vez asignamos el widget Home() de tipo StatefulWidget que es la parte funcional de nuestra aplicación y que retornará a fin de cuentas un Scaffold() que es el widget que contiene finalmente toda la estructura de nuestra interfaz.

**// --- Estructura estándar de un Scaffold**

....

return Scaffold(

appBar: , // --- barra de título

body: , // --- contenido principal

bottomNavigationBar: , // --- navigación

floatingActionButton: // --- botón de acción

)

...

## Hola Mundo en Flutter

En la propiedad home será donde anidaremos los widgets relacionados con el scaffold de nuestra aplicación. Recordemos que el scaffold contiene la estructura base para una aplicación en Material Design.

La composición básica del scaffold para nuestro Hola Mundo será la siguiente:

...

MaterialApp (

...

home: Scaffold (

appBar: AppBar(

title: Text('Hola Mundo')

),

body: Center(

child: Text('HolaMundo')

),

)

)

Además del appBar y el body, un scaffold estándar también puede incluir un bottomNavigationVar y un floatingActionButton, entre otros widgets.

Es importante tener en cuenta que un error común al definir la estructura de widgets para nuestra aplicación es olvidar colocar el widget correspondiente al Scaffold(). Esto se hace más evidente cuando al previsualizar nuestra app, vemos que los widgets de tipo Text() aparecen con unas rayas amarillas en la parte inferior.

En Flutter, al hacer cambios en el código y guardar el archivo, se actualiza automáticamente la interfaz del emulador, compilando solo la zona modificada, lo que nos permite ver los cambios reflejados casi de manera instantánea.

## Widgets básicos

““En Flutter, casi todo es un widget””. Son los componentes elementales de la interfaz de usuario de la aplicación.

Los widgets básicos son el punto de partida con los que puedes comenzar a crear tus propios widgets (compuestos o más complejos) en Flutter.

Text: representa un texto o cadena de caracteres. Admite diversos parámetros o propiedades, además del texto literal que será mostrado. Podemos definirle estilos y características de apariencia particulares.

Row: permite organizar y posicionar otros widgets hijos en forma de fila o renglón.

Column: permite organizar y posicionar otros widgets hijos en forma de columna.

Stack: permite apilar widgets hijos con respecto al eje Z (elevación) en la interfaz.

Container: es uno de los widgets más versátiles de Flutter y permite organizar otros widgets como si fuera una caja o contenedor. Este widget es equivalente a los divs de html y admiten atributos similares de apariencia.

## Widgets con estado y sin estado

Flutter dispone principalmente de dos tipos de Widgets:

### Widgets con estado o StatefulWidgets

Son todos los widgets con los que el usuario de la aplicación puede tener una interacción directa. Por ejemplo: Checkbox, Radio, Slider, Form. Este tipo de widgets por lo general disparan algún evento, acción o comportamiento determinado como respuesta a la interacción con el usuario.

### Widgets sin estado o StatelessWidgets

Los widgets de este tipo no reaccionan ante el intento de interacción con el usuario. No tienen asociado una acción o comportamiento particular. Son estáticos en la interfaz. Por ejemplo: Icono, Texto, Contendor con color, entre otros.

Ambos tipos de widgets heredan sus características y comportamientos de una clase principal Widget.

## Análisis de Interfaces de Usuario en Flutter

Analizar las interfaces de usuario para determinar la mejor forma de componer los diferentes grupos de elementos visuales o de interacción que conforman nuestra aplicación es una tarea común entre desarrolladores de aplicaciones móviles. De hecho la propia documentación oficial de Flutter lo recomienda.

El análisis consiste básicamente en identificar o detectar visualmente cuál sería la mejor forma de construir las diferentes secciones de nuestra interfaz, tomando en cuenta los widgets básicos de los cuales disponemos en Flutter. Prever la manera en que pueden ser anidados u organizados los diferentes widgets básicos, incluso determinar qué grupos de widgets pudieran pasar a ser nuevos widgets compuestos nos facilitará mucho el trabajo posterior ya cuando pasemos al código del programa.

Una técnica usada para representar el widget tree resultante del análisis de la interfaz es mediante un diagrama de árbol, en el que cada widget se representa con una símbolo circular en una estructura de dependencias o asociaciones derivadas desplegadas en forma de árbol descendente.

# 4. Widgets sin estado en Flutter

## Flutter Widgets: Container, Text, Icon, Row

Ahora que ya tenemos el análisis de la interfaz de nuestro proyecto, conoceremos el primer grupo de widgets que nos ofrece Flutter y con el que iremos dando forma a nuestra aplicación.

Container: como su nombre lo indica, es un contenedor que nos permite agrupar u organizar internamente otros widgets.

Text: se utiliza para mostrar textos simples.

Icon: se utiliza para incluir iconos predefinidos en la interfaz.

Row: es un contenedor en el que los elementos internos se organizan de forma horizontal y seguidos unos de otros.

Cada uno de los widgets tienen propiedades, algunas comunes y otras particulares que iremos viendo a lo largo del curso.

Como ya hemos visto en clases anteriores: ““En Flutter, prácticamente TODO es un widget””, así que la forma de pasar del análisis de la interfaz al código es mediante la composición de widgets propios (o clases) que contendrán los diferentes grupos anidados de widgets básicos para cada sección de la interfaz. De esta manera tendremos una estructura mucho más organizada y mantenible.

Los nombres de archivo en Dart, con Flutter, generalmente deben ir escritos en minúsculas y usando underscore en lugar de espacios, a este estilo se le llama SnakeCase; sin embargo, internamente, los nombres de las clases se escribirán con mayúscula inicial o estilo PascalCase.

La primera línea del archivo de clase en Dart que define un Widget, es la siguiente:

import 'package:flutter/material.dart';

Y con ella se importan al contexto global de nuestra aplicación todas las definiciones de widgets que vienen predefinidos y listo para usarse con Flutter.

## Flutter Widgets: Column

El widget Column, como ya vimos, nos permite organizar distintos elementos visuales de la interfaz (también widgets) de manera vertical, alineados uno arriba / debajo del otro en el eje Y.

Los widgets hijos, que van a ser organizados dentro de un Column padre, deben estar definidos en la propiedad children de éste, como un arreglo de elementos de tipo <Widget>[...].

Algo que debemos recordar es que los widgets que usamos como children, pueden definirse bien sea directamente (inline) o bien a través de clases externas (o widgets personalizados) que hemos definido previamente en archivos .dart independientes.

Para utilizar valores que nos permitan tener contenido dinámico en nuestra interfaz es necesario apoyarnos en el método constructor que lleva el mismo nombre de la clase y en el que definimos los parámetros de entrada que serán utilizados en lugar de las variables por defecto que también hemos definido antes.

## Recursos en Flutter: Tipografías y Google Fonts

Cuando hablamos de Tipografías nos referimos precisamente al tipo y estilo de letras que usaremos en los textos de nuestra interfaz, esta característica estará determinada por las indicaciones que nos haya suministrado nuestro equipo de diseño.

Un sitio web muy utilizado e interesante para obtener tipografías que puedes usar en tus proyectos es Google Fonts.

Para incorporar las tipografías personalizadas a nuestro proyecto crearemos una carpeta particular llamada fonts/ en la que guardaremos los archivos que descarguemos desde Google Fonts. Seguidamente, debemos declarar este recurso en el archivo pubspecs.yaml. Este archivo contiene las especificaciones de todos los recursos públicos que estarán disponibles para ser usados en nuestro proyecto. La sintaxis yaml está inspirada en json y xml y se usa en Flutter para definir la estructura de recursos y configuraciones globales de nuestra aplicación.

La forma de declarar los recursos de tipografía que usaremos en nuestro proyecto es la siguiente:

...

fonts:

- family: <nombre de la fuente>

fonts:

- asset: <ubicación del archivo de fuente>

...

Algo que se debe tener en cuenta es que yaml es bastante estricto con la indentación de los datos. En otras palabras, la sangría, tabulación o espaciado de cada línea debe estar indicada correctamente para que no haya problemas con la lectura y procesamiento posterior de las configuraciones.

Para aplicar la nueva tipografía a nuestro código bastará con incluir en el widget TextStyle, la propiedad fontFamily e indicar en ella el nombre de la fuente que hemos declarado en nuestro archivo de configuración.

## Widget Image

Similar a la manera en que declaramos e incorporamos los recursos de tipografía en la clase anterior, las imágenes al ser también recursos externos, deberán ser guardadas en un directorio particular que llamaremos assets/ y declaradas en el archivo pubspecs.yaml de la siguiente manera:

...

assets:

- <ubicación del recurso en la estructura de archivos del proyecto>

...

Una vez declarado el recurso de imagen en el archivo de configuraciones, lo incorporamos al código mediante el widget AssetImage( “<ubicación del asset>” ) que a su vez colocaremos como valor de la propiedad image de un widget DecorationImage, o de cualquier otro widget que disponga de la propiedad image, como es el caso del BoxDecoration.

## Widget Apilando Textos

Cuando hablamos de apilar textos nos referimos a organizarlos de tal forma que queden unos arriba de otros formando items compuestos que luego serán parte de una lista. Para esto nos apoyaremos en los widgets: Container(), Text() y Column(), dispuestos convenientemente en un widget tree que en próximas clases completaremos con la funcionalidad de scrolling como ya veremos.

Siguiendo el análisis de la interfaz de nuestro proyecto, que hicimos en clases anteriores, haremos la composición de nuestro widget review, y luego del widget reviewList.

Presta especial atención al uso de los widgets Text y TextStyle, ya que son claves para definir la apariencia de los textos en la lista de reviews.

Progresivamente iremos apilante los textos que vamos definiendo de forma individual, usando el widget Column(), que como ya hemos visto en clases anteriores, nos permite organizar los widgets hijos unos arriba de los otros en el eje Y.

## Widgets Decorados

Al hablar de widgets decorados nos referimos básicamente a un widget de tipo Container(), al que podemos darle características visuales extendidas, o de decoración. Esto lo podemos conseguir al asignar a la propiedad decoration de dicho Container(), otro widget especial llamado BoxDecoration().

Algunas de las propiedades de decoración disponibles con este widget son:

color: para asignar el color de relleno de fondo del contenedor.

gradient: para definir un gradiente de colores de relleno.

border: para asignar un borde.

borderRadius: para indicar que los bordes del contenedor serán redondeados.

## Widget Imagen Decorada

Como ya hemos visto en clases anteriores, al momento de incorporar imágenes a nuestra interfaz, basta con utilizar el widget AssetImage() como valor de la propiedad image en un BoxDecoration(), Container(), o similar; habiendo declarado previamente las imágenes como assets en el archivo de configuraciones pubspecs.yaml de Flutter.

Ahora bien, si queremos dar a esas imágenes características visuales o de decoración particulares, como: alineación, repetición en mosaico, filtros de color o ajuste de tamaño (fit), entre otras, será necesario disponer entonces de un widget diferente: el DecorationImage(), el cual usaremos en sustitución del tradicional AssetImage(). De esta manera tendremos disponibles todas estas propiedades especiales, además de la propiedad que ya conocemos: image que sí, nuevamente, tendría que estar definida por un AssetImage().

TIP: para evitar que el statusBar de Android se vea con un color más oscuro, habitual en este SO, podemos agregar al final de la clase MainActivity del archivo MainActivity.kt (ubicado en la carpeta src/main/kotlin/), la siguiente línea de código:

this.getWindow().statusBarColor(android.graphics.Color.TRANSPARENT);

## Widget Listview

La particularidad del widget ListView(), como ya lo habíamos adelantado en clases anteriores, es la capacidad de permitir el desplazamiento del contenido (generalmente items de una lista) que en principio queda ubicado fuera de la región visible de nuestra interfaz.

En esta clase implementamos el ListView(), con la particularidad de que el desplazamiento o scrolling que definiremos en la propiedad scrollDirection será horizontal y no vertical para lo cual usaremos la variable Axis.horizontal.

Algo importante que vale la pena recordar es que con Flutter podemos crear widgets personalizados que luego podemos parametrizar y reutilizar tantas veces como queramos.

## Widget Button, InkWell

El BottomNavigationBar es un widget muy común para la navegación principal a través de las diferentes pantallas o vistas de nuestra aplicación. Al igual que el FloatingActionButton(), y como responderá a la interacción con el usuario, al BottomNavigationBar() le corresponde heredar de la clase StatefulWidget.

Para definir las características de la apariencia que tendrá este widget, lo envolveremos a su vez en un widget de tipo Theme() y le asignaremos finalmente el widget BottomNavigationBar() a la propiedad child.

# 5. Widgets con estado en Flutter

## Clase StatefulWidget: Cómo se compone

Flutter maneja fundamentalmente dos clases de Widgets, los que heredan (o extienden) sus propiedades y comportamientos a partir de la clase StatelessWidget y los que heredan (o extienden) de la clase StatefulWidget. Comprender la diferencia entre estos dos tipos de Widgets será especialmente importante cuando vayamos a programar la lógica de nuestra aplicación.

Lo primero que debemos comprender es qué significan los términos Stateless y Stateful:

Stateless: sin estado. (Que es inmutable. Que no cambia)

Stateful: que posee estado. (Que puede cambiar)

El estado es un concepto propio de la programación funcional, y en términos muy generales se refiere al valor o situación de todo el conjunto de variables, métodos y funciones contenidas en una clase en un momento dado. Para ilustrarlo un poco, sería como una “fotografía” de todos los valores asociados a una clase. Lo peculiar de este concepto es que un estado puede ser mutable, que puede cambiar o ser cambiado en algún momento durante la ejecución de la aplicación, dependiendo de algún evento o variable externa; o, por otro lado, pudiera ser inmutable, que nunca cambia, o que sus valores no dependen de ningún elemento o evento externo o interacción con el usuario.

En síntesis, el State es un tipo de clase particular que almacena la situación, o el estado, de un Widget durante la ejecución de la aplicación y, como parte de su comportamiento, presta atención a los eventos o interacciones externas que le sucedan a éste y que puedan generar una modificación o alteración de sus propiedades, métodos o funciones. Como resultado, el State devuelve una nueva “fotografía”, con la nueva situación ya cambiada del Widget y este se vuelve a renderizar en la interfaz.

## Widget Floating Action Button

Como vimos en la clase anterior, para que un widget sea capaz de responder a la interacción con el usuario y en función de esa interacción pueda cambiar su comportamiento o algunas de sus propiedades, este debe ser definido inicialmente como un StatefulWidget.

En esta clase aprenderemos a construir, personalizar y darle comportamiento a un Floating Action Button, nuestro widget de tipo Statful, o con estado.

Este widget, también conocido como Fab, es muy común en las interfaces móviles basadas en Material Design y generalmente representan al elemento de principal interacción esperada en la interfaz, equivale a un Call to action.

La sintaxis básica de un Stateful Widget, es:

import 'package:flutter/material.dart';

class <nombre del widget> extends StatefulWidget {

@override

State<StatefulWidget> createState() {

return \_<nombre del widget>();