
Creación de una aplicación Hello World

Una excelente manera de aprender un nuevo lenguaje de desarrollo es escribir una aplicación básica. En programación, Hello World es el programa más básico para escribir. Simplemente muestra las palabras Hello World en la pantalla. En este tema, aprenderás los pasos principales para desarrollar este programa básico como una aplicación Flutter. No te preocupes todavía por comprender el código; Te guiaremos paso a paso más adelante en esta parte del curso.

Escribir este ejemplo mínimo te ayuda a aprender la estructura básica de una aplicación Flutter, cómo ejecutar la aplicación en el simulador de iOS y el emulador de Android, y cómo realizar cambios en el código.

Configurar el proyecto

La configuración inicial del proyecto para cada aplicación es la misma. Estamos usando Android Studio para crear las aplicaciones de ejemplo en esta parte del curso, pero puedes elegir un editor diferente, como IntelliJ o Visual Studio Code.

Una descripción general del proceso en Android Studio es la siguiente: crea un nuevo proyecto de Flutter, selecciona la aplicación Flutter como el tipo de proyecto (plantilla) e ingresa el nombre del proyecto. Luego, el kit de desarrollo de software (SDK) de Flutter crea el proyecto por ti, incluida la creación de un directorio de proyecto con el mismo nombre que el nombre del proyecto.

Dentro del directorio del proyecto, la carpeta lib contiene el archivo main.dart con el código fuente (en otras palabras, `project_name/lib/main.dart`). También tendrás una carpeta de Android para la aplicación de Android, la carpeta de ios para la aplicación de iOS y una carpeta de prueba para las pruebas unitarias.

En este tema, la carpeta lib y el archivo main.dart son tu enfoque principal. En el tema 4, “Creación de una plantilla de proyecto de inicio”, aprenderás cómo crear un proyecto de inicio de Flutter y cómo estructurar el código en archivos separados.

De forma predeterminada, la aplicación Flutter usa widgets de componentes de materiales basados en Material Design. Material Design es un sistema de pautas de mejores prácticas para el diseño de interfaces de usuario. Los componentes de un proyecto de Flutter incluyen widgets visuales, de comportamiento y de movimiento. Los proyectos de Flutter también

incluyen pruebas unitarias para widgets, que son archivos que contienen código individual para probar si la lógica funciona según lo diseñado.

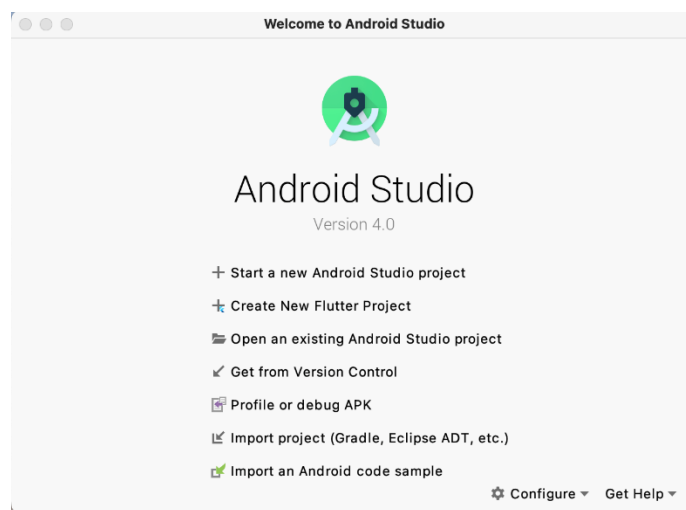
Las aplicaciones de Flutter se crean con el lenguaje de programación Dart, y aprenderás los conceptos básicos de Dart en el tema 3, “Aprendizaje de los conceptos básicos de Dart”.

Creando una nueva aplicación

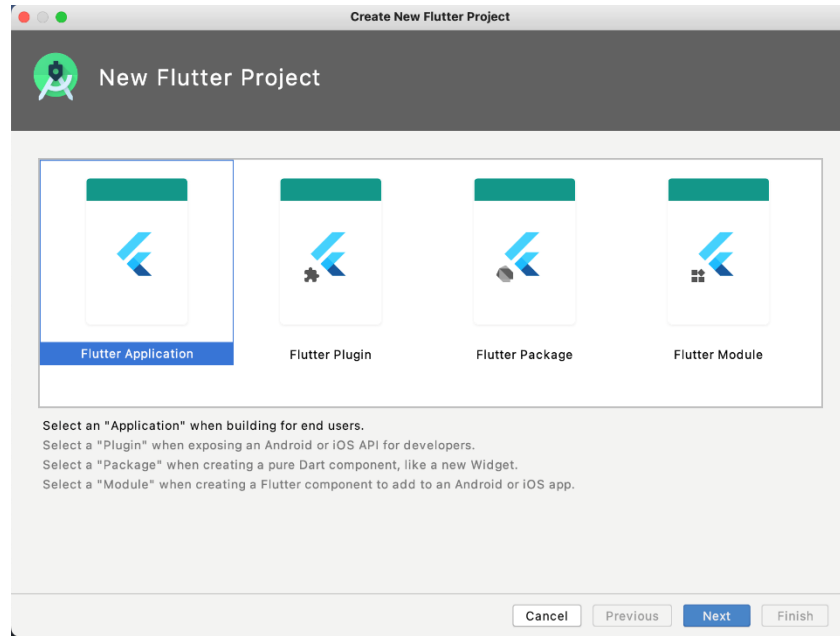
En este ejercicio, crearás una nueva aplicación Flutter llamada `my_counter`. El nombre de la aplicación es el mismo que el del proyecto. Esta aplicación utiliza la plantilla de proyecto Flutter mínima predeterminada e incluye un botón de acción flotante (el botón `+`) que aparece en la parte inferior derecha de la pantalla del dispositivo. Cada vez que se toca este botón, un contador aumenta en uno.

La plantilla Flutter actual crea la aplicación básica que tiene un contador y un botón `+` (el botón de acción flotante). En futuras versiones de Flutter, es posible que haya otras opciones de plantilla disponibles. ¿Por qué el equipo de Flutter decidió usar un contador como plantilla? Es inteligente porque muestra cómo tomar datos, manipularlos y mantener el estado (el valor del contador) con recarga en caliente. Se puede realizar en la versión 4.0 de Android.

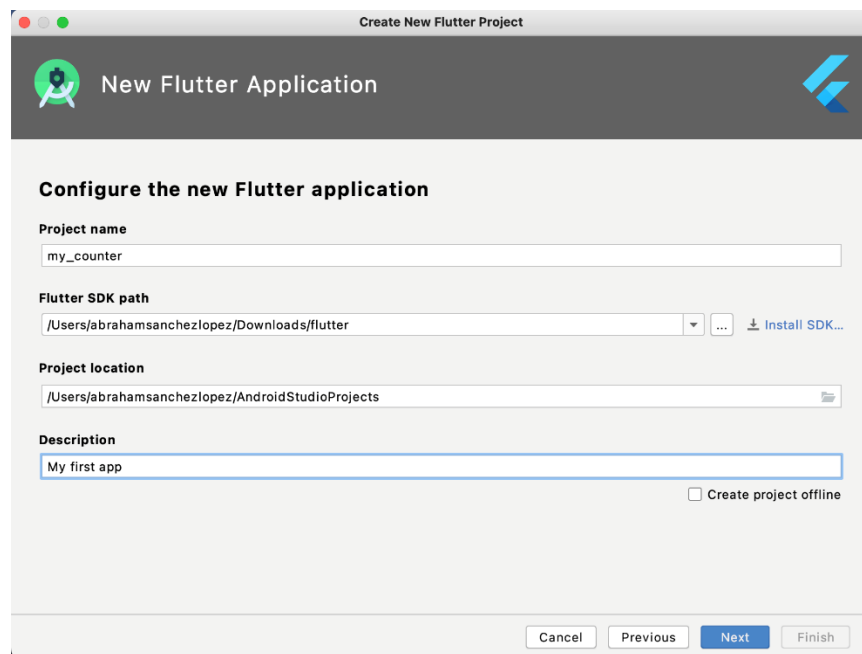
1. Inicia Android Studio.
2. Si tienes un proyecto abierto, haz clic en la barra de menú y selecciona `File ⇄ New ⇄ New Flutter Project`. Si no hay ningún proyecto abierto, haz clic en `Iniciar un nuevo proyecto de Flutter`.



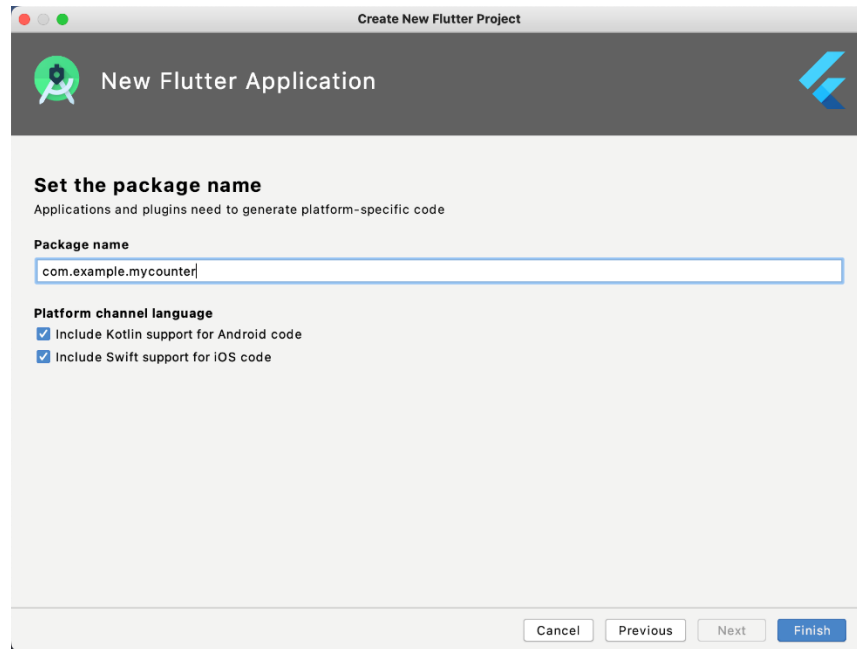
3. Selecciona Aplicación Flutter.



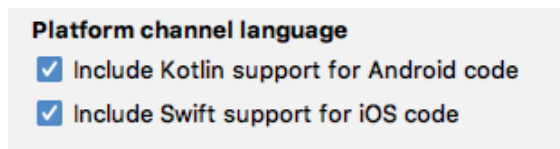
4. La convención de nombrado habitual es separar las palabras con un guión bajo, así que ingresa el nombre del proyecto `my_counter` y haz clic en Siguiente (Next). Ten en cuenta que la ruta del SDK de Flutter es la carpeta de instalación que elegiste en tus proyectos de Android. Opcionalmente, puedes cambiar la ubicación y descripción del proyecto.



5. Ingresa el nombre de tu empresa en el campo dominio de la empresa y formateálo como nombre de dominio.com. Utiliza cualquier nombre exclusivo si no tienes un nombre de empresa.



6. En la misma pantalla que el paso 5 para el lenguaje del canal de la plataforma (Platform Channel Language), selecciona las opciones para Kotlin y Swift.



Estas opciones asegurarán que estés utilizando los últimos lenguajes de programación tanto para Android (Kotlin) como para iOS (Swift). El uso del canal de la plataforma, una forma de comunicarse entre el código Dart de la aplicación y el código específico de la plataforma, te permite usar interfaces de programación de aplicaciones (API) específicas de la plataforma iOS y Android escribiendo código en cada lenguaje nativo, como escribir código nativo en Kotlin (Android) y Swift (iOS) para manejar la reproducción de audio mientras la aplicación está en modo de fondo. Verás más de cerca los canales de la plataforma, Kotlin y Swift en el tema 12, “Escritura de código nativo de la plataforma”.

7. Haz clic en el botón Finalizar.

Cómo funciona

Se crea un proyecto de Flutter con el nombre de carpeta `my_counter`, el mismo que el nombre del proyecto. Utiliza la plantilla de proyecto estándar de Flutter, que muestra cómo actualizar un contador tocando el botón `+`.

De forma predeterminada, se utilizan los componentes de Material Design de Android. En un proyecto de Flutter, estos incluyen widgets visuales, de comportamiento y de movimiento. Para cada proyecto que crees, hay un directorio llamado `lib`, y el archivo `main.dart` se ejecutará primero cuando se ejecute la aplicación. El archivo `main.dart` contiene código Dart con la función `main()` que inicia la aplicación. El objetivo de crear esta aplicación es familiarizarte con cómo se crea y estructura una aplicación Flutter.

El tema 3 cubre los conceptos básicos de Dart, y el tema 4 cubre el archivo `main.dart` con más detalle, particularmente cómo estructura y separa la lógica del código.

Utilizando hot reload

La recarga en caliente (hot reload) de Flutter te ayuda a ver el código y los cambios en la interfaz de usuario de inmediato mientras retienes el estado de una aplicación que ejecuta la máquina virtual Dart. En otras palabras, cada vez que realizas cambios en el código, no necesitas volver a cargar la aplicación, porque la página actual muestra los cambios de inmediato. Esta es una característica increíble para ahorrar tiempo a cualquier desarrollador.

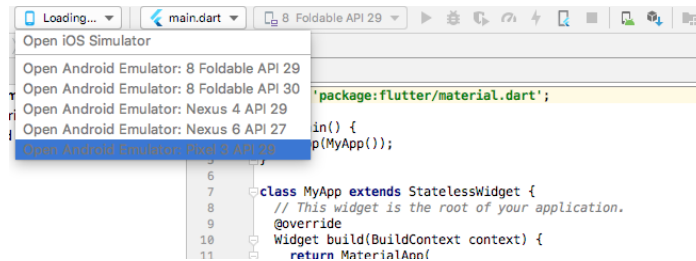
En Flutter, la clase `State` almacena datos mutables (modificables). Por ejemplo, la aplicación comienza con el valor del contador establecido en cero, pero cada vez que tocas el botón `+`, el valor del contador aumenta en uno.

Cuando se toca el botón `+` tres veces, el valor del contador muestra un valor de tres. El valor del contador es mutable, por lo que puede cambiar con el tiempo. Al usar la recarga en caliente, puedes realizar cambios en la lógica del código y el estado del contador de la aplicación (valor) no se restablece a cero, sino que retiene el valor actual de tres.

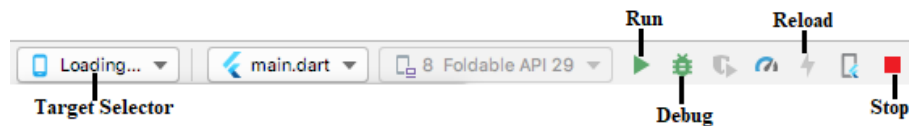
Ejecutando la aplicación

Para ver cómo funciona la recarga en caliente, iniciarás el emulador/simulador, realizarás cambios en el título de la página, los guardarás y verás que los cambios ocurren de inmediato. Desde el tema 1, el simulador de iOS se creó automáticamente cuando instalaste Xcode, y creaste manualmente el emulador de Android. El simulador de iOS solo está disponible si se ejecuta Android Studio en una computadora Mac porque requieres la instalación de Xcode de Apple. Se supone que tienes instalados el simulador de iOS y el emulador de Android. Si no es así, usa el emulador de Android.

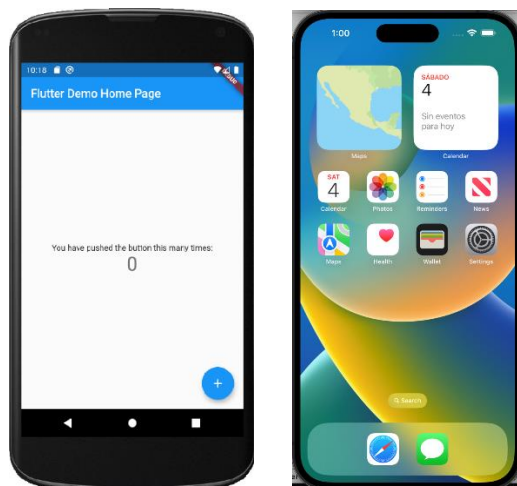
1. Desde Android Studio, haz clic en el botón de selección de dispositivo Flutter a la derecha de la barra de herramientas. Una lista desplegable muestra el simulador de iOS y el emulador de Android disponibles.





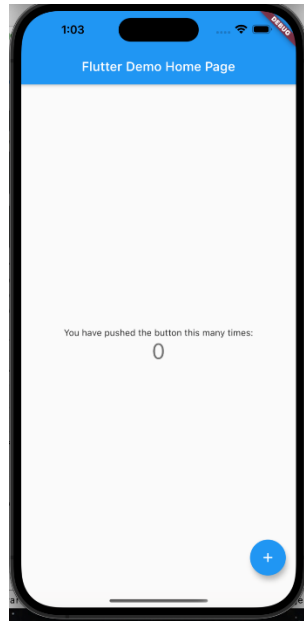
2. Selecciona el simulador de iOS o el emulador de Android.
3. Haz clic en el icono Ejecutar en la barra de herramientas.



4. Deberías ver la aplicación my_counter en el simulador. Sigue el paso 2 para ejecutar la aplicación tanto en el simulador de iOS (si está disponible) como en el emulador de Android. Al ejecutar la aplicación tanto en iOS como en Android, es evidente que tienen el mismo aspecto, pero hereda los rasgos de cada sistema operativo móvil. Ten en cuenta en el simulador de iOS que el título de la aplicación está centrado, pero en el emulador de Android, el título está a la izquierda.
5. Haz clic en el botón de acción flotante + en la parte inferior derecha y verás que el contador aumenta cada vez que hace clic en él.



6. En el archivo main.dart, cambia MyHomePage(title: 'Flutter Demo Home Page') a MyHomePage(title: 'Hello World') y guarda presionando  (en Windows ). Inmediatamente verás que el título de la barra de la aplicación cambia y el estado del contador sigue siendo el mismo, sin restablecer a cero. Este cambio instantáneo se denomina recarga en caliente y lo usarás con frecuencia para mejorar tu productividad.



Cómo funciona

La recarga en caliente es una característica increíble que ahorra tiempo para ver los resultados de los cambios en el código fuente de inmediato mientras se mantiene el estado actual. Mientras la máquina virtual Dart se está ejecutando, la recarga en caliente inyecta el código fuente actualizado y el framework Flutter reconstruye el árbol de widgets. (El árbol de widgets se tratará en detalle en el tema 5, “Comprensión del árbol de widgets”).

Uso de temas para estilizar tu aplicación

Los widgets de tema son una excelente manera de diseñar y definir colores globales y estilos de fuente para su aplicación.



Hay dos formas de utilizar los widgets de temas: para diseñar la apariencia y el estilo de forma global o para diseñar solo una parte de la aplicación. Por ejemplo, puedes usar temas para diseñar el brillo del color (texto claro sobre un fondo oscuro o viceversa); los colores primarios y de acento; el color del lienzo; y el color de las barras de aplicaciones, tarjetas, divisores, opciones seleccionadas y no seleccionadas, botones, sugerencias, errores, texto, iconos, etc.

La belleza de Flutter es que la mayoría de los elementos son widgets y casi todo es personalizable. De hecho, personalizar la clase ThemeData te permite cambiar el color y la tipografía de los widgets. (Aprenderás más sobre los widgets en detalle en el tema 5, “Comprensión del árbol de widgets” y el tema 6, “Uso de widgets comunes”).

Uso de un tema de aplicación global

Tomemos la nueva aplicación my_counter y modifiquemos el color primario. El color actual es azul, así que vamos a cambiarlo a verde claro. Agrega una nueva línea debajo de primarySwatch y agrega código para cambiar el color de fondo (canvasColor) a lightGreen.

```
primarySwatch: Colors.blue,  
// Change it to  
primarySwatch: Colors.lightGreen,  
canvasColor: Colors.lightGreen.shade100,
```

Guarda presionando  (en Windows ). Se invoca la recarga en caliente, por lo que la barra de aplicaciones y el lienzo ahora son de un tono verde claro.

Para mostrar un poco de genialidad de Flutter, agrega una propiedad de plataforma de TargetPlatform.iOS después de la propiedad canvasColor y ejecuta la aplicación desde el emulador de Android. De repente, los rasgos de iOS se están ejecutando en Android. El título de la barra de la aplicación no se alinea a la izquierda, sino que se cambia al centro, que es el estilo habitual de iOS (figura 2.1).

```
primarySwatch: Colors.blue,  
// Change it to  
primarySwatch: Colors.lightGreen,  
canvasColor: Colors.lightGreen.shade100,  
platform: TargetPlatform.iOS
```

Esto se puede hacer a la inversa mediante TargetPlatform. Específicamente, para mostrar los rasgos de Android en iOS, cambia la propiedad de la plataforma a TargetPlatform.android y ejecuta la aplicación desde el simulador de iOS. El título de la barra de la aplicación no está alineado al centro, pero ha cambiado para estar alineado a la izquierda, que es el estilo habitual de Android (figura 2.2). Una vez que se implemente la navegación con varias páginas, esto será aún más evidente. En iOS, cuando navegas a una nueva página, generalmente desliza la página siguiente desde el lado derecho de la pantalla hacia la izquierda. En Android, cuando navegas a una nueva página, generalmente desliza la página siguiente de abajo hacia arriba. TargetPlatform tiene tres opciones: Android, Fuchsia (el sistema operativo en desarrollo de Google) e iOS.


```

24
25 // this makes the visual density adapt to the platform that you run
26 // the app on. For desktop platforms, the controls will be smaller and
27 // closer together (more dense) than on mobile platforms.
28 primarySwatch: Colors.lightGreen,
29 canvasColor: Colors.lightGreen.shade100,
30 platform: TargetPlatform.android,
31 visualDensity: VisualDensity.adaptivePlatformDensity,
32 }, // ThemeData
33 home: MyHomePage(title: 'Hello World'),
34 ); // MaterialApp
35 }

```

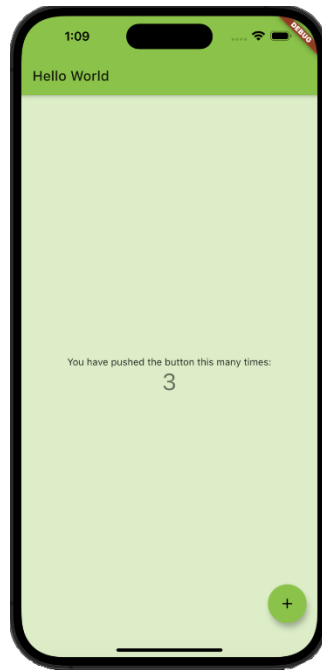


Figura 2.1: Rasgos de iOS que se ejecutan en Android.

Aquí hay otro ejemplo del uso de un tema de aplicación global: si tienes un banner de depuración rojo en la parte superior derecha del emulador, puedes desactivarlo con el siguiente código. Google tenía la intención de informar a los desarrolladores que el rendimiento de la aplicación no está en modo de lanzamiento. Flutter crea una versión de depuración de la aplicación y el rendimiento es más lento. Usando el modo de liberación (solo dispositivo), crea una aplicación con optimización de velocidad.

Agrega la propiedad `debugShowCheckedModeBanner` y establece el valor en falso, y se eliminará el banner de depuración rojo. Desactivar el banner rojo de depuración es solo por motivos estéticos; todavía estás ejecutando una versión de depuración de la aplicación.

```

return new MaterialApp(
  debugShowCheckedModeBanner: false,
  title: 'My Counter',
  theme: new ThemeData(
    ...

```

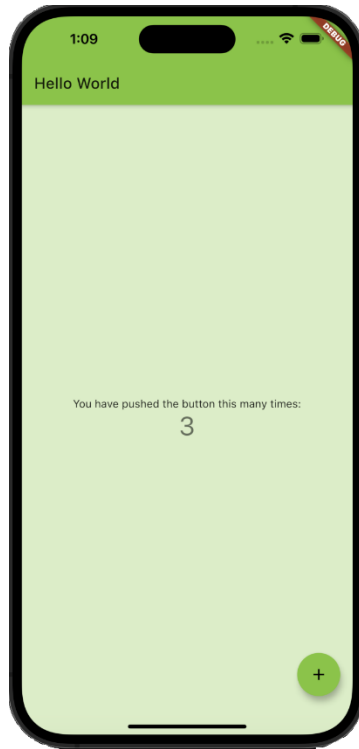


Figura 2.2: Rasgos de Android que se ejecutan en iOS.

Uso de un tema para parte de una aplicación

Para anular el tema de toda la aplicación, puedes envolver widgets en un widget de tema. Este método anulará completamente la instancia de ThemeData de la aplicación sin heredar ningún estilo.

En la sección anterior, cambiaste primarySwatch y canvasColor a lightGreen, lo que afectó a todos los widgets de la aplicación. ¿Qué sucede si solo deseas que un widget en una página tenga un esquema de color diferente y que el resto de los widgets use el tema de la aplicación global predeterminado? Anula el tema predeterminado con un widget de tema que usa la propiedad de datos para personalizar ThemeData (como cardColor, primaryColor y canvasColor), y el widget secundario usa la propiedad de datos para personalizar los colores.

```
body: Center(
  child: Theme(
    // Unique theme with ThemeData – Overwrite
    data: ThemeData(
      cardColor: Colors.deepOrange,
    ),
    child: Card( child: Text('Unique ThemeData'),
    ),
  ),
),
```

Recomendamos extender el tema principal de la aplicación, cambiando solo las propiedades necesarias y heredando el resto. Usa el método `copyWith` para crear una copia del tema principal de la aplicación y reemplaza solo las propiedades que necesitas cambiar. Al desglosarlo, `Theme.of(context).copyWith()` extiende el tema principal y puedes anular las propiedades necesarias dentro de `copyWith(cardColor: Colors.DeepOrange)`.

```
body: Center(
  child: Theme(
    // copyWith Theme - Inherit (Extended)
    data: Theme.of(context).copyWith(cardColor: Colors.deepOrange),
    child: Card(
      child: Text('copyWith Theme'),
    ),
  ),
),
```

El siguiente código de muestra ilustra cómo cambiar el color predeterminado de la tarjeta a `deepOrange` con el tema sobrescrito (`ThemeData()`) y extendido (`Theme.of()` `.CopyWith()`) para producir el mismo resultado.

Los dos widgets de tema se envuelven dentro de un widget de columna para alinearlos verticalmente. En este punto, no te preocupes por el widget `Column`, ya que se trata en el tema 6.

```
body: Column(
  children: <Widget>[
    Theme(
      // Unique theme with ThemeData – Overwrite
      data: ThemeData(
        cardColor: Colors.deepOrange,
      ),
      child: Card(
        child: Text('Unique ThemeData'),
      ),
    ),
    Theme(
      // copyWith Theme - Inherit (Extended)
      data: Theme.of(context).copyWith(cardColor: Colors.deepOrange),
      child: Card(
        child: Text('copyWith Theme'),
      ),
    ),
  ],
),
```

Entendiendo widgets stateless y stateful

Los widgets de Flutter son los componentes básicos para diseñar la interfaz de usuario (UI). Los widgets se crean utilizando un framework moderno de estilo react. La interfaz de usuario se crea anidando los widgets en un árbol de widgets.

El framework de estilo react de Flutter significa que observa cuándo cambia el estado de un widget y luego lo compara con el estado anterior para determinar la menor cantidad de cambios a realizar. Flutter administra la relación entre el estado y la interfaz de usuario y reconstruye solo esos widgets cuando cambia el estado.

En esta sección, compararás widgets sin estado (stateless) y con estado (stateful) y aprenderás cómo implementar cada clase y, lo que es más importante, cuál usar según el requerimiento. En temas posteriores, crearás aplicaciones para cada escenario. La clase apropiada se amplía (convirtiéndola en una subclase) mediante el uso de la palabra clave *extends* seguida de `StatelessWidget` o `StatefulWidget`.

`StatelessWidget` se utiliza cuando los datos no cambian y se basa en la información inicial. Es un widget sin estado y los valores son definitivos. Algunos ejemplos son `Text`, `Button`, `Icon` e `Image`.

```
class Instructions extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Text('When using a StatelessWidget...');  
  }  
}
```

`StatefulWidget` se usa cuando los datos cambian. Es un widget con un estado que puede cambiar con el tiempo y requiere dos clases. Para que los cambios se propaguen a la interfaz de usuario, es necesario realizar una llamada al método `setState()`.

- Clase `StatefulWidget`: Crea una instancia de la clase `State`.
- Clase `State`: Esto es para datos que se pueden leer sincrónicamente cuando se crea el widget y pueden cambiar con el tiempo.
- `setState ()`: Desde dentro de la clase `State`, realiza una llamada al método `setState()` para actualizar los datos modificados, indicándole al framework que el widget debe volver a dibujarse porque el estado ha cambiado. Para todas las variables que necesitan cambios, modifica los valores en `setState () {_ myValue += 50.0;})`. Cualquier valor de variable modificado fuera del método `setState()` no actualizará la interfaz de usuario. Por lo tanto, es mejor colocar los cálculos que no necesitan cambios de estado fuera del método `setState()`.

Considera el ejemplo de una página que muestra tu oferta máxima en un producto. Cada vez que se presiona el botón Aumentar oferta (Increase Bid), tu oferta aumenta en \$50. Empiezas por crear una clase `MaximumBid` que extienda la clase `StatefulWidget`. Creas una clase `_MaximumBidState` que extienda el estado de la clase `MaximumBid`.

En la clase `_MaximumBidState`, declaras una variable llamada `_maxBid`. El método `_increaseMyMaxBid()` llama al método `setState()`, que incrementa el valor `_maxBid` en \$50. La interfaz de usuario consta de un widget de texto que muestra el valor 'My Maximum Bid: \$_maxBid' y un `FlatButton` con una propiedad `onPressed` que llama al método `_increaseMyMaxBid()`. El método `_increaseMyMaxBid()` ejecuta el método `setState()`, que agrega \$50 a la variable `_maxBid`, y la cantidad del widget `Text` se vuelve a dibujar.

```
class MaximumBid extends StatefulWidget {
  @override
  _MaximumBidState createState() => _MaximumBidState();
}
```

```
class _MaximumBidState extends State<MaximumBid> {
  double _maxBid = 0.0;

  void _increaseMyMaxBid() {
    setState(() {
      // Add $50 to my current bid
      _maxBid += 50.0;
    });
  }
}
```

```
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      Text('My Maximum Bid: $_maxBid'),
      FlatButton.icon(
        onPressed: () => _increaseMyMaxBid(),
        icon: Icon(Icons.add_circle),
        label: Text('Increase Bid'),
      ),
    ],
  );
}
```

Utilizar paquetes externos

A veces no vale la pena crear un widget desde cero. Flutter admite paquetes de terceros para los ecosistemas Flutter y Dart. Los paquetes contienen la lógica del código fuente y se comparten fácilmente. Hay dos tipos de paquetes, Dart y plugin.

- Los paquetes de Dart están escritos en Dart y pueden contener dependencias específicas de Flutter.
- Los paquetes de complementos están escritos en Dart (con el código de Dart exponiendo la API) pero se combinan con implementaciones de código específicas de la plataforma para Android (Java o Kotlin) y/o iOS (Objective-C o Swift). La mayoría de los paquetes de complementos tienen como objetivo admitir tanto Android como iOS.

Supongamos que buscas agregar funcionalidad a tu aplicación, como mostrar algunos gráficos, acceder a las ubicaciones de GPS de un dispositivo, reproducir audio de fondo o acceder a una base de datos como Firebase. Hay paquetes para todo eso.

Buscando paquetes

En la aplicación, digamos que necesitas almacenar las preferencias del usuario tanto en iOS como en Android y deseas encontrar un paquete que lo haga por ti.

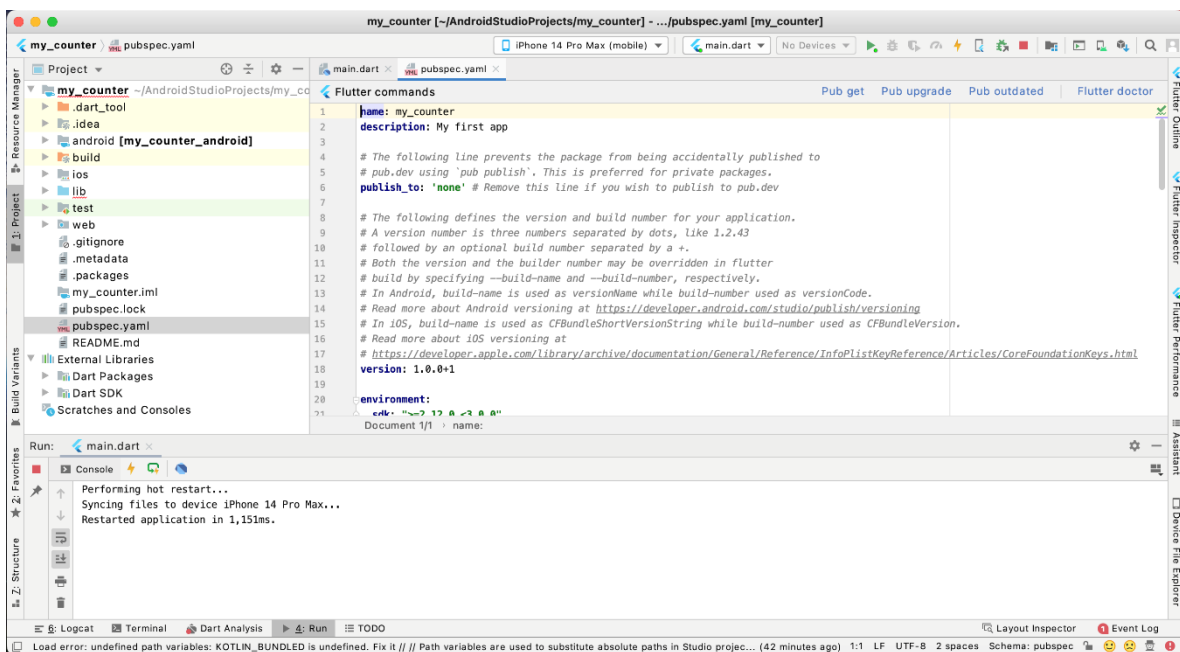
1. Inicia tu navegador web y navega hasta <https://pub.dev/flutter/packages>. Los paquetes son publicados en esta ubicación a menudo por otros desarrolladores y Google.
2. Haz clic en la barra de búsqueda Buscar paquetes de Flutter. Ingresa las preferencias compartidas y los resultados se ordenarán por relevancia.
3. Haz clic en el enlace del paquete `shared_preferences`. (El enlace directo es https://pub.dev/packages/shared_preferences).
4. Los detalles sobre cómo instalar y utilizar el paquete `shared_preferences` están disponibles en esta ubicación. El equipo de Flutter es el autor de este paquete en particular. Haz clic en la pestaña Instalación para obtener instrucciones detalladas. Cada paquete tiene instrucciones sobre cómo instalarlo y usarlo. La instalación de la mayoría de los paquetes es similar, pero difieren en cómo usar e implementar el código. Las instrucciones se encuentran en la página de inicio de cada paquete.

Instalación de paquetes

Has aprendido a buscar paquetes de terceros. A continuación, aprenderás a implementar el paquete externo `shared_preferences` en tu aplicación.

1. Abre la aplicación my_counter con Android Studio.
2. Abre el archivo pubspec.yaml haciendo doble clic.
3. En dependencias :, section, agrega shared_preferences: ^ 0.5.1 + 1. (Tu versión puede ser superior).
4. Guarda el archivo y se instalará el paquete. Si no ves que el proceso se ejecuta automáticamente, puedes invocarlo manualmente ingresando los paquetes de flutter en la ventana de tu Terminal en Android Studio. Una vez terminado, el mensaje dirá Proceso terminado con código de salida 0.
5. Importa el paquete en el archivo main.dart después de la línea de importación material.dart, ubicada en la parte superior del archivo. Guarde los cambios.

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



Cómo funciona

Al agregar las dependencias shared_preference en el archivo pubspec.yaml, las dependencias se descargan en el proyecto local. Utiliza la declaración de importación para que el paquete shared_preference esté disponible.

Usar paquetes

Cada paquete tiene su forma única de implementarse. Siempre es bueno leer la documentación.

Para el paquete `shared_preferences`, debes agregar algunas líneas para implementarlo. Recuerda que el punto principal aquí no es cómo usar este paquete, sino cómo agregar paquetes externos a tu aplicación en general.

Implementar e inicializar un paquete

En la clase `_MyHomePageState`, agrega una función llamada `_updateSharedPreferences()`.

```
class _MyHomePageState extends State<MyHomePage> {  
  // ...  
  void _updateSharedPreferences() async {  
    SharedPreferences prefs = await SharedPreferences.getInstance();  
    int counter = (prefs.getInt('counter') ?? 0) + 1;  
    print('Pressed $counter times.');
```

Cómo funciona

Este paquete guarda las preferencias de los usuarios tanto en iOS como en Android con unas pocas líneas de código Dart, que es extremadamente poderoso. No es necesario escribir código nativo para iOS o Android. Este es el poder de usar paquetes, pero ten cuidado de no exagerar porque depende de los autores de los paquetes para mantenerlos actualizados.

Resumen

En este segundo tema, aprendiste cómo crear tu primera aplicación y usar la recarga en caliente para ver los cambios instantáneamente. También viste cómo usar temas para diseñar aplicaciones, cuándo usar widgets sin estado y con estado, y cómo agregar paquetes externos para evitar reinventar la rueda.

Ahora tienes una comprensión general de las ideas principales detrás del desarrollo de aplicaciones Flutter. No te preocupes todavía por comprender el código real. Lo aprenderás todo a lo largo del curso.

En el próximo tema, aprenderás los conceptos básicos del lenguaje Dart que se usa para crear aplicaciones Flutter.