Usando los widgets comunes

En este nuevo tema, aprenderás a usar los widgets más comunes. Los llamamos nuestros bloques de construcción base para crear hermosas UI y UX. Aprenderás a cargar imágenes localmente o en la Web a través de un localizador de recursos uniforme (URL), usar los íconos ricos de componentes de materiales incluidos y aplicar decoradores para mejorar la apariencia de los widgets o usarlos como guías de entrada para los campos de entrada.

También explorarás cómo aprovechar el widget de formulario (Form) para validar los widgets de entrada de campos de texto como grupo, no solo individualmente. Además, para tener en cuenta la variedad de tamaños de dispositivos, verás cómo el uso del widget MediaQuery o OrientationBuilder es una excelente manera de detectar la orientación porque es extremadamente importante usar los widgets de orientación y diseño del dispositivo según sea portrait o landscape. Por ejemplo, si el dispositivo está en modo vertical, puedes mostrar una fila de tres imágenes, pero cuando el dispositivo está en modo horizontal, puedes mostrar una fila de cinco imágenes ya que el ancho es un área más grande que en el modo vertical.

Usar widgets básicos

Al crear una aplicación móvil, normalmente implementarás ciertos widgets para la estructura base. Es necesario estar familiarizado con ellos.

Scaffold. Como aprendiste en el tema 4, el widget Scaffold implementa el diseño visual básico de Material Design, lo que te permite agregar fácilmente varios widgets como AppBar, BottomAppBar, FloatingActionButton, Drawer, SnackBar, BottomSheet y más.

AppBar. El widget AppBar generalmente contiene el título estándar, la barra de herramientas, las propiedades iniciales y de acciones (junto con los botones), así como muchas opciones de personalización.

> *title*. La propiedad del título se implementa normalmente con un widget de texto. Puedes personalizarlo con otros widgets, como un widget DropdownButton.
>
> *leading*. La propiedad principal se muestra antes de la propiedad del título. Por lo general, es un IconButton o BackButton.

actions. La propiedad de acciones se muestra a la derecha de la propiedad del título. Es una lista de widgets alineados en la parte superior derecha de un widget de AppBar, generalmente con un IconButton o PopupMenuButton.
flexibleSpace. La propiedad flexibleSpace se apila detrás de la barra de herramientas o el widget TabBar. La altura suele ser la misma que la del widget AppBar. Normalmente se aplica una imagen de fondo a la propiedad flexibleSpace, pero se puede utilizar cualquier widget, como un icono.

SafeArea. El widget SafeArea es necesario para los dispositivos actuales, como el iPhone X o los dispositivos Android con una muesca (un corte parcial que oculta la pantalla que generalmente se encuentra en la parte superior del dispositivo). El widget SafeArea agrega automáticamente suficiente relleno al widget secundario para evitar intrusiones por parte del sistema operativo. Opcionalmente, puede pasar una cantidad mínima de relleno (padding) o un valor booleano para no aplicar el relleno en la parte superior, inferior, izquierda o derecha.

Container. El widget de contenedor es un widget de uso común que permite la personalización de su widget hijo. Puedes agregar fácilmente propiedades como color, ancho, alto, relleno, margen, borde, restricción, alineación, transformación (como rotar o cambiar el tamaño del widget) y muchas otras. La propiedad secundaria es opcional y el widget Container se puede usar como un marcador de posición vacío (invisible) para agregar espacio entre widgets.

Text. El widget de texto se utiliza para mostrar una cadena de caracteres. El constructor de Text toma los argumentos string, style, maxLines, overflow, textAlign y otros. Un constructor es cómo se pasan los argumentos para inicializar y personalizar el widget de texto.

RichText. El widget RichText es una excelente manera de mostrar texto usando múltiples estilos. El widget RichText toma TextSpans como elementos secundarios para diseñar diferentes partes de las cadenas.

Column. Un widget de columna muestra sus elementos secundarios verticalmente. Se necesita una propiedad secundaria que contenga una matriz de List<Widget>, lo que significa que puedes agregar varios widgets. Los hijos se alinean verticalmente sin ocupar toda la altura de la pantalla. Cada widget secundario (child) se puede incrustar en un widget expandido para llenar el espacio disponible. CrossAxisAlignment, MainAxisAlignment y MainAxisSize se pueden usar para alinear y dimensionar la cantidad de espacio ocupado en el eje principal.

Row. Un widget de fila muestra sus hijos horizontalmente. Toma una propiedad secundaria (children) que contiene una matriz de List <Widget>. Las mismas propiedades que contiene

la columna se aplican al widget Row con la excepción de que la alineación es horizontal, no vertical.

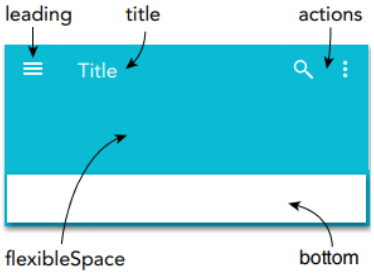
Buttons. Hay una variedad de botones para elegir para diferentes situaciones como RaisedButton, FloatingActionButton, FlatButton, IconButton, PopupMenuButton y ButtonBar.

Agregar widgets de AppBar

Crea un nuevo proyecto Flutter y asígnale el nombre basics; puedes seguir las instrucciones del tema 4. Para este proyecto, solo necesitas crear la carpeta de páginas (pages). El objetivo de esta aplicación es proporcionar una idea de cómo usar los widgets básicos, no necesariamente para diseñar la interfaz de usuario más atractiva. Más adelante, nos centraremos en crear diseños complejos y hermosos.

1. Abre el archivo main.dart. Cambia la propiedad primarySwatch de blue a lightGreen.
   primarySwatch: Colors.lightGreen,

2. Abre el archivo home.dart. Empieza por personalizar las propiedades del widget AppBar.



3. Agrega a la AppBar un leading IconButton. Si anulas la propiedad leading, suele ser un IconButton o BackButton.

```
leading: IconButton(
  icon: Icon(Icons.menu),
  onPressed: () { },
),
```

4. La propiedad del title suele ser un widget de texto, pero se puede personalizar con otros widgets como DropdownButton. Siguiendo las instrucciones del tema 4, ya has agregado el widget de Text a la propiedad del título; si no es así, agrega el widget de texto con un valor de 'Home'.

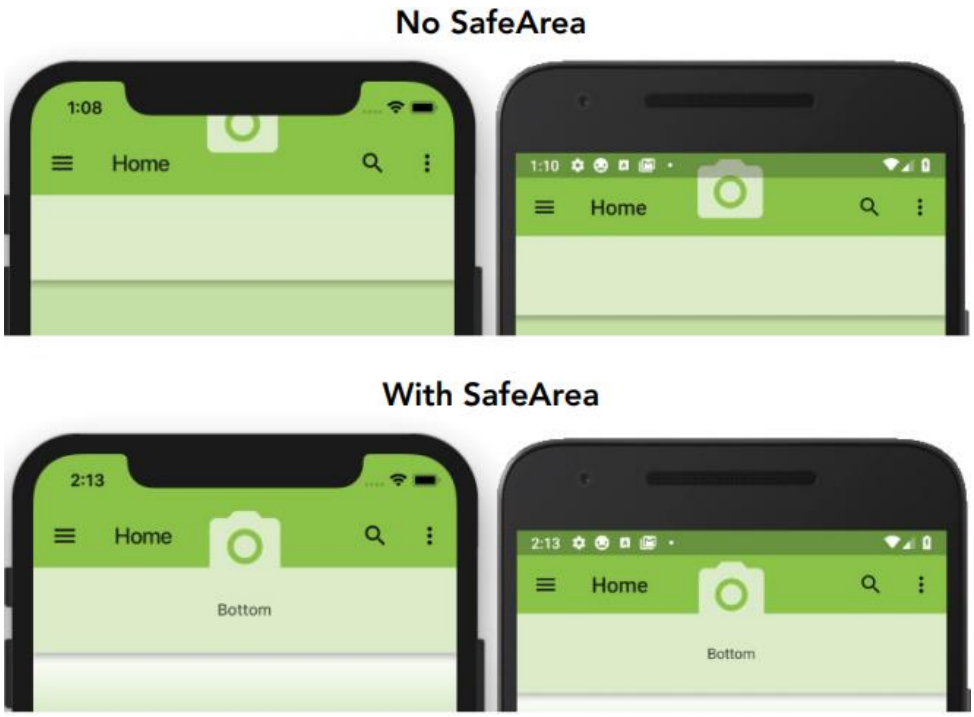
```
title: Text('Home'),
```

5. La propiedad de acciones toma una lista de widgets; agrega dos widgets IconButton.

```
actions: <Widget>[
  IconButton(
    icon: Icon(Icons.search),
    onPressed: () {},
  ),
  IconButton(
    icon: Icon(Icons.more_vert),
    onPressed: () {},
  ),
],
```

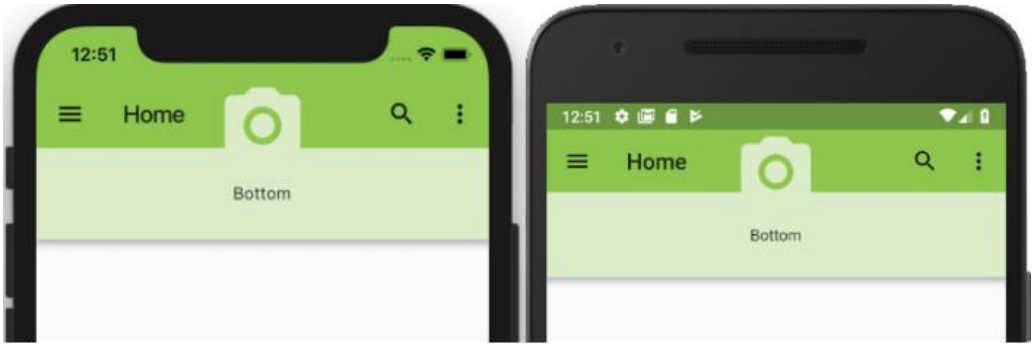
6. Dado que estás utilizando un icono para la propiedad flexibleSpace, agreguemos un SafeArea y un icono como child.

```
flexibleSpace: SafeArea(
  child: Icon(
    Icons.photo_camera,
    size: 75.0,
    color: Colors.white70,
  ),
),
```



7. Agrega un PreferredSize para la propiedad bottom con un Container para un child.

```
bottom: PreferredSize(
   child: Container(
      color: Colors.lightGreen.shade100,
      height: 75.0,
      width: double.infinity,
      child: Center(
         child: Text('Bottom'),
      ),
   ),
   preferredSize: Size.fromHeight(75.0),
),
```



Aprendiste a personalizar el widget AppBar mediante el uso de widgets para configurar el título (title), la barra de herramientas (toolbar), las propiedades iniciales (leading) y las acciones (actions). Todas las propiedades que aprendiste en este ejemplo están relacionadas con la personalización de AppBar.

En el tema 9, “Creación de listas de desplazamiento y efectos”, aprenderás a utilizar el widget SliverAppBar, que es una AppBar incrustada en una astilla (sliver) mediante CustomScrollView, lo que hace que cualquier aplicación cobre vida con personalizaciones precisas como la animación de paralaje. Nos encanta usar slivers porque agregan una capa adicional de personalización.

En la siguiente sección, aprenderás a personalizar la propiedad del Scaffold body anidando widgets para crear el contenido de la página.

Area segura (SafeArea)

El widget SafeArea es imprescindible para los dispositivos actuales, como el iPhone X o los dispositivos Android con una muesca (notch) (un corte parcial que oscurece la pantalla que generalmente se encuentra en la parte superior del dispositivo). El widget SafeArea agrega automáticamente suficiente relleno al widget secundario para evitar intrusiones por parte del sistema operativo. Opcionalmente, puedes pasar un relleno mínimo o un valor booleano para no aplicar el relleno en la parte superior, inferior, izquierda o derecha.

Agregar un SafeArea al cuerpo

Continua modificando el archivo home.dart.

Agrega un widget de relleno (Padding) a la propiedad del cuerpo (body) con SafeArea como hijo (child). Debido a que este ejemplo incluye diferentes usos de widgets, agrega un SingleChildScrollView como un elemento secundario (child) de SafeArea. SingleChildScrollView permite al usuario desplazarse y ver widgets ocultos; de lo contrario, el usuario ve una barra amarilla y negra que indica que los widgets están desbordados.

```
body: Padding(
   padding: EdgeInsets.all(16.0),
   child: SafeArea(
     child: SingleChildScrollView(
       child: Column(
         children: <Widget>[

         ],
       ),
     ),
   ),
),
```

Ejemplo de sugerencia de widget de envoltura

Existe una excelente manera de ajustar un widget actual como hijo de otro widget. Coloca el cursor en la parte superior del widget actual para ajustar y luego presiona Option + Enter en tu teclado. Aparece el Dart/quick de asistencia rápida. Elije la opción Wrap con un nuevo widget.

No agregues los siguientes pasos a tu proyecto; este es un consejo sobre cómo envolver (wrap) rápidamente un widget con otro.

1. Coloca el cursor en el widget para ajustar (wrap).

2. Presiona Option + Enter (Alt + Enter en Windows). Aparece el Dart/quick de asistencia rápida.

3. Selecciona una opción Wrap con nuevo widget como body: widget (child: Container()),.

4. Cambia el nombre del widget a SafeArea y observa que child: es automáticamente el widget Container(). Asegúrate de agregar una coma después del widget Container(), como se muestra aquí. Colocar una coma después de cada propiedad asegura el formato correcto de Flutter en varias líneas.

   body: SafeArea(child: Container(),),

La adición del widget SafeArea ajusta automáticamente el relleno (padding) para los dispositivos que tienen una muesca. Todos los widgets secundarios (child) de SafeArea están restringidos al relleno correcto.

Container

El widget Container tiene una propiedad de widget secundario (child) opcional y se puede usar como un widget decorado con un borde, color, restricción, alineación, transformación (como girar el widget) personalizados y más.
Este widget se puede utilizar como un marcador de posición vacío (invisible) y, si se omite un child, se ajusta al tamaño de pantalla completo disponible.

Agregar un contenedor

Continua modificando el archivo home.dart. Dado que deseas mantener tu código legible y manejable, crearás clases de widgets para construir cada sección de widgets del cuerpo de la lista de widgets de Column.

1. Agrega a la propiedad del cuerpo un widget de relleno con la propiedad secundaria establecida en un widget SafeArea. Agrega al child SafeArea un

SingleChildScrollView. Agrega al elemento secundario SingleChildScrollView una columna. Para los hijos de Column, agrega la llamada a la clase de widget ContainerWithBoxDecorationWidget(), que crearás a continuación. Asegúrate de que la clase de widget usa la palabra clave const para aprovechar el almacenamiento en caché (rendimiento).

```
body: Padding(
   padding: EdgeInsets.all(16.0),
   child: SafeArea(
      child: SingleChildScrollView(
        child: Column(
          children: <Widget>[
             const ContainerWithBoxDecorationWidget(),
          ],
        ),
      ),
   ),
),
```

2. Crea la clase de widget ContainerWithBoxDecorationWidget() después de que la clase Home amplíe StatelessWidget {...}. La clase de widget devolverá un widget. Ten en cuenta que cuando refactorizas creando clases de widgets, estas son de tipo StatelessWidget a menos que especifiques usar un StatefulWidget.

```
class ContainerWithBoxDecorationWidget extends StatelessWidget {
  const ContainerWithBoxDecorationWidget({
    Key key,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
      return Column(
        children: <Widget>[
           Container(),
        ],
      );
  }
}
```

3. Comienza a agregar propiedades al Container agregando una altura de 175.0 píxeles. Ten en cuenta la coma después del número, que separa las propiedades y ayuda a mantener el código de Dart formateado. Ve a la siguiente línea para agregar la propiedad de decoración, que acepta una clase BoxDecoration. La clase

BoxDecoration proporciona diferentes formas de dibujar un cuadro, y en este caso, estás agregando una clase BorderRadius a bottomLeft y bottomRight del Container.

```
Container(
    height: 100.0,
    decoration: BoxDecoration(),
),
```

4. El uso del constructor denominado BorderRadius.only() te permite controlar los lados para dibujar esquinas redondeadas. A propósito, hicimos el radio bottomLeft mucho más grande que el bottomRight para mostrar las formas personalizadas que puedes crear.

```
BoxDecoration(
  borderRadius: BorderRadius.only(
    bottomLeft: Radius.circular(100.0),
    bottomRight: Radius.circular(10.0),
  ),
),
```

BoxDecoration también admite una propiedad de degradado. Estás utilizando un LinearGradient, pero también podrías haber usado un RadialGradient. El LinearGradient muestra los colores del degradado linealmente y el RadialGradient muestra los colores del degradado de forma circular. Las propiedades de inicio (begin) y finalización (end) te permiten elegir las posiciones de inicio y finalización del degradado mediante la clase AlignmentGeometry.
AlignmentGeometry es una clase base para Alignment que permite una resolución orientada a la dirección. Tienes muchas direcciones para elegir, como Alignment.bottomLeft, Alignment.centerRight y más.

```
begin: Alignment.topCenter,
end: Alignment.bottomCenter,
```

La propiedad de colores requiere una Lista de tipos de color, List <Color>. La lista de colores se ingresa entre corchetes separados por comas.

```
colors: [
  Colors.white,
  Colors.lightGreen.shade500,
],
```

Aquí está el código fuente de la propiedad de gradiente completo:

```
gradient: LinearGradient(
  begin: Alignment.topCenter,
```

```
    end: Alignment.bottomCenter,
    colors: [
        Colors.white,
        Colors.lightGreen.shade500,
    ],
),
```

5. La propiedad boxShadow es una excelente manera de personalizar una sombra y toma una lista de BoxShadows, llamada List<BoxShadow>. Para BoxShadow, establece las propiedades de color, blurRadius y offset.

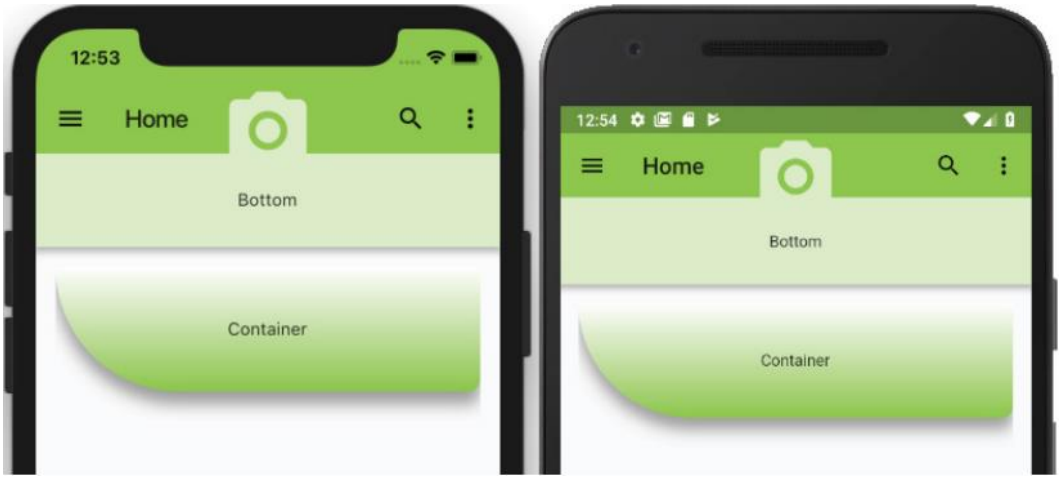
```
boxShadow: [
    BoxShadow(
        color: Colors.white,
        blurRadius: 10.0,
        offset: Offset(0.0, 10.0),
    )
],
```

La última parte del Container es agregar un widget de texto secundario envuelto por un widget de centro. El widget del Centro te permite centrar el widget del child en la pantalla.

```
child: Center(
    child: Text('Container'),
 ),
```

6. Agrega un widget Center como hijo del contenedor y agrega al widget Center hijo de un widget Text con la cadena Container. (En la siguiente sección, repasaremos el widget de texto en detalle).

La siguiente figura (página 11) muestra el código fuente completo de la clase del widget ContainerWithBoxDecorationWidget().




```
class ContainerWithBoxDecorationWidget extends StatelessWidget {
  const ContainerWithBoxDecorationWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Container(
          height: 100.0,
          decoration: BoxDecoration(
            borderRadius: BorderRadius.only(
              bottomLeft: Radius.circular(100.0),
              bottomRight: Radius.circular(10.0),
            ),
            gradient: LinearGradient(
              begin: Alignment.topCenter,
              end: Alignment.bottomCenter,
              colors: [
                Colors.white,
                Colors.lightGreen.shade500,
              ],
            ),
            boxShadow: [
              BoxShadow(
                color: Colors.grey,
                blurRadius: 10.0,
                offset: Offset(0.0, 10.0),
              ),
            ],
          ),
          child: Center(
            child: RichText(
              text: Text('Container'),          ),
        ),
      ],
    );
  }
}
```

Los contenedores pueden ser widgets poderosos llenos de personalización. Mediante el uso de decoradores, degradados y sombras, puedes crear hermosas interfaces de usuario. Me gusta pensar que los contenedores mejoran una aplicación de la misma manera que un marco de gran apariencia agrega a una pintura.

Texto (Text)

Ya usaste el widget de texto en los ejemplos anteriores; es un widget fácil de usar pero también personalizable. El constructor de Text toma los argumentos string, style, maxLines, overflow, textAlign y otros.

```
Text(
  'Flutter World for Mobile',
  style: TextStyle(
    fontSize: 24.0,
    color: Colors.deepPurple,
    decoration: TextDecoration.underline,
    decorationColor: Colors.deepPurpleAccent,
    decorationStyle: TextDecorationStyle.dotted,
    fontStyle: FontStyle.italic,
    fontWeight: FontWeight.bold,
  ),
  maxLines: 4,
  overflow: TextOverflow.ellipsis,
  textAlign: TextAlign.justify,
),
```

Texto rico (RichText)

El widget RichText es una excelente manera de mostrar texto usando múltiples estilos. El widget RichText toma TextSpan como elementos secundarios para diseñar diferentes partes de las cadenas (figura 6.1).

Figura 6.1: RichText con TextSpan.

Reemplazo de texto con un contenedor secundario RichText

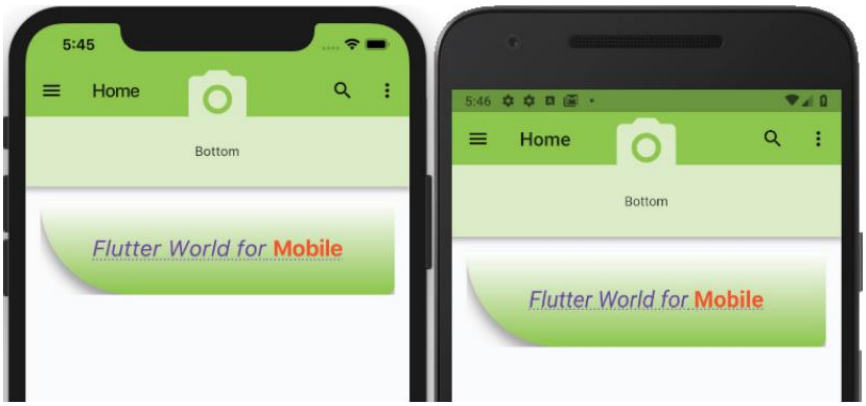
En lugar de usar el widget Container Text anterior para mostrar una propiedad de texto sin formato, puedes usar un widget RichText para mejorar y enfatizar las palabras en tu cadena. Puedes cambiar el color y el estilo de cada palabra.

1. Busca el widget de texto secundario (child) del Container y elimina el texto ('Container').

```
child: Center(
  child: Text('Container'),
),
```

2. Reemplaza el widget Text del hijo Container con un widget RichText. La propiedad de texto RichText es un objeto TextSpan (clase) que se personaliza mediante un TextStyle para la propiedad de estilo. TextSpace tiene una lista secundaria de TextSpan donde colocas diferentes objetos TextSpan para dar formato a diferentes partes de todo el RichText.
   Al utilizar el widget RichText y combinar diferentes objetos TextSpan, creas un formato de texto enriquecido como con un procesador de texto.

```
child: Center(
  child: RichText(
    text: TextSpan(
      text: 'Flutter World',
      style: TextStyle(
        fontSize: 24.0,
        color: Colors.deepPurple,
        decoration: TextDecoration.underline,
        decorationColor: Colors.deepPurpleAccent,
        decorationStyle: TextDecorationStyle.dotted,
        fontStyle: FontStyle.italic,
        fontWeight: FontWeight.normal,
      ),
      children: <TextSpan>[
        TextSpan(
          text: ' for',
        ),
        TextSpan(
          text: ' Mobile',
          style: TextStyle(
              color: Colors.deepOrange,
              fontStyle: FontStyle.normal,
              fontWeight: FontWeight.bold),
        ),
      ],
    ),
  ),
),
```



RichText es un widget poderoso cuando se combina con el objeto TextSpan (clase). Hay dos partes principales del estilo, la propiedad de texto predeterminada y la lista secundaria de TextSpan.

La propiedad de texto que usa TextSpan establece el estilo predeterminado para RichText. La lista de hijos de TextSpan te permite usar múltiples objetos TextSpan para formatear diferentes cadenas.

Columna (Column)

Un widget de columna (figuras 6.2 y 6.3) muestra sus elementos secundarios (children) verticalmente. Toma una propiedad secundaria que contiene una matriz de List<Widget>.

Los children se alinean verticalmente sin ocupar toda la altura de la pantalla. Cada widget secundario se puede incrustar en un widget expandido para llenar el espacio disponible. Puedes utilizar CrossAxisAlignment, MainAxisAlignment y MainAxisSize para alinear y dimensionar la cantidad de espacio ocupado en el eje principal.

```
Column(
  crossAxisAlignment: CrossAxisAlignment.center,
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Text('Column 1'),
    Divider(),
    Text('Column 2'),
    Divider(),
    Text('Column 3'),
  ],
),
```

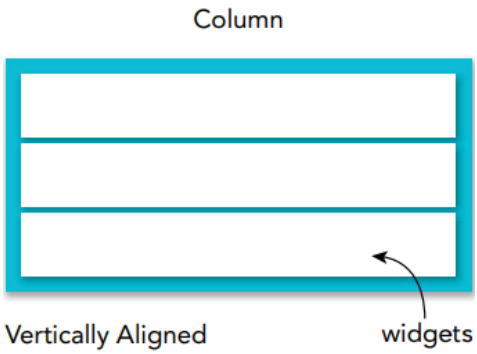


Figura 6.2: Widget de columna.

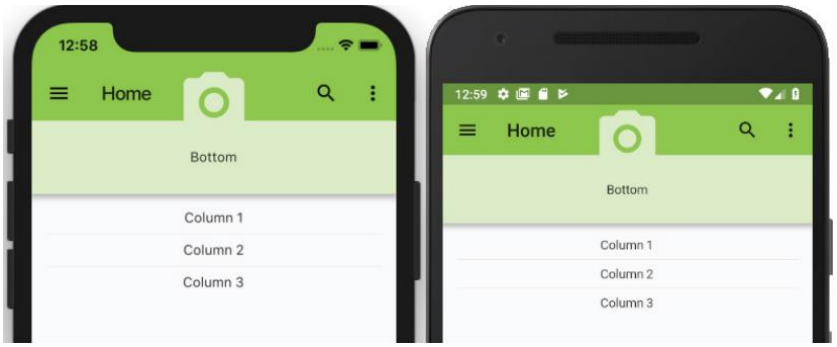


Figura 6.3: Widget de columna renderizado en la aplicación.

Fila (Row)

Un widget Fila (figuras 6.4 y 6.5) muestra sus hijos horizontalmente. Toma una propiedad secundaria que contiene una matriz de List<Widget>. Las mismas propiedades que contiene la columna se aplican al widget Fila con la excepción de que la alineación es horizontal, no vertical.

```
Row(
  crossAxisAlignment: CrossAxisAlignment.start,
  mainAxisAlignment: MainAxisAlignment.
spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Row(
      children: <Widget>[
        Text('Row 1'),
        Padding(padding: EdgeInsets.all(16.0),),
        Text('Row 2'),
        Padding(padding: EdgeInsets.all(16.0),),
        Text('Row 3'),
      ],
    ),
  ],
),
```

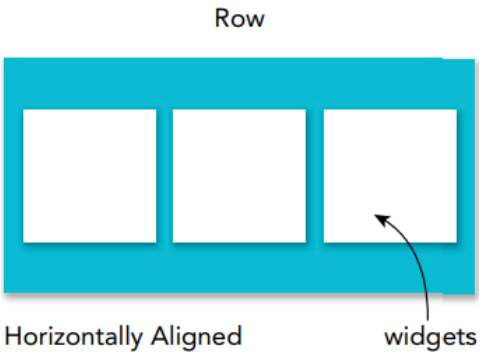


Figura 6.4: Widget de fila.

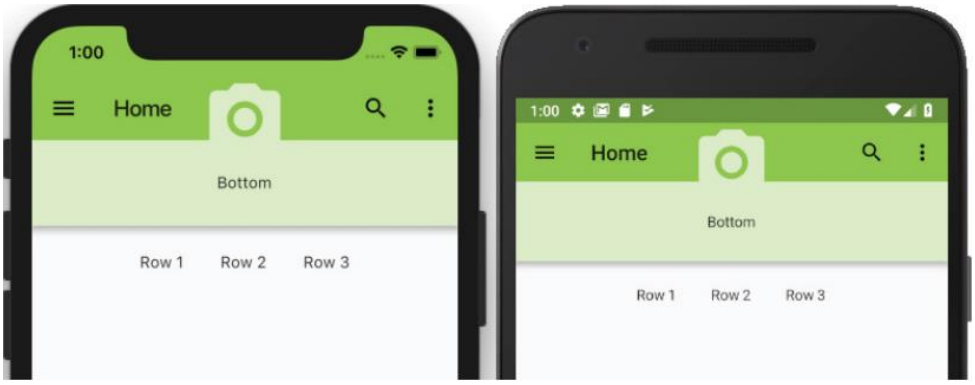


Figura 6.5: Widget de fila renderizado en la aplicación.

Anidación de columnas y filas

Una excelente manera de crear diseños únicos es combinar los widgets de columna y fila para satisfacer las necesidades individuales. Imagínate tener una página de un diario con texto en una columna con una fila anidada que contiene una lista de imágenes (figuras 6.6 y 6.7).

```
Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  mainAxisAlignment: MainAxisAlignment.
spaceEvenly,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Text('Columns and Row Nesting 1',),
    Text('Columns and Row Nesting 2',),
    Text('Columns and Row Nesting 3',),
    Padding(padding: EdgeInsets.all(16.0),),
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: <Widget>[
        Text('Row Nesting 1'),
        Text('Row Nesting 2'),
        Text('Row Nesting 3'),
      ],
    ),
  ],
),
```

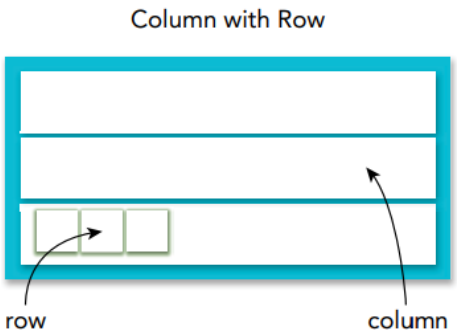


Figura 6.6: Anidación de columnas y filas.

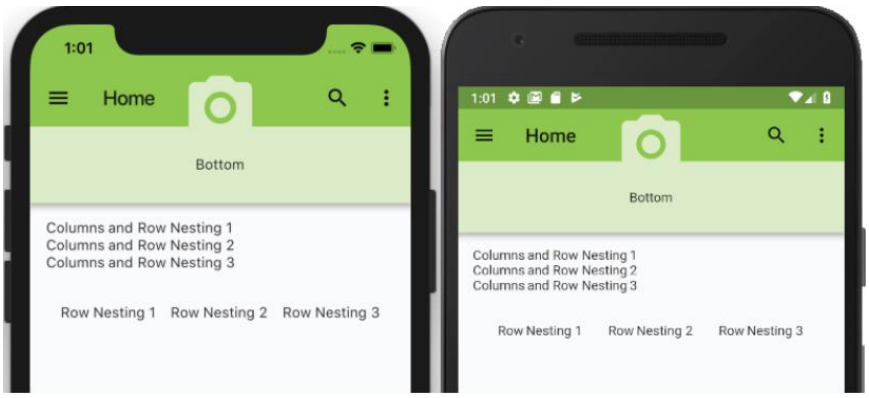


Figura 6.7: Widgets de columna y fila renderizados en la aplicación.

Agregar columna, fila y anidar la fila y la columna juntas como clases de widget

Agregarás tres clases de widgets a la sección de propiedades del cuerpo de la lista Columna de widgets. Entre cada clase de widget, agregarás un widget Divider() simple para dibujar líneas de separación entre las secciones.

1. Agrega los nombres de clase de widget ColumnWidget(), RowWidget() y ColumnAndRowNestingWidget() a la lista de widgets secundarios de Column. El widget Columna se encuentra en la propiedad del cuerpo. Agregue un widget Divider() entre cada nombre de clase de widget. Asegúrate de que cada clase de widget use la palabra clave const.

```
body: Padding(
  padding: EdgeInsets.all(16.0),
  child: SafeArea(
    child: SingleChildScrollView(
      child: Column(
        children: <Widget>[
          const ContainerWithBoxDecorationWidget(),
          Divider(),
          const ColumnWidget(),
          Divider(),
          const RowWidget(),
          Divider(),
         const ColumnAndRowNestingWidget(),
        ],
      ),
    ),
  ),
),
```

2. Crea la clase de widget ColumnWidget() después de la clase de widget ContainerWithBoxDecorationWidget().

```
class ColumnWidget extends StatelessWidget {
  const ColumnWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.center,
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      mainAxisSize: MainAxisSize.max,
      children: <Widget>[
        Text('Column 1'),
        Divider(),
        Text('Column 2'),
        Divider(),
        Text('Column 3'),
      ],
    );
  }
}
```

3. Crea la clase de widget RowWidget() después de la clase de widget ColumnWidget().

```
class RowWidget extends StatelessWidget {
  const RowWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      crossAxisAlignment: CrossAxisAlignment.start,
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      mainAxisSize: MainAxisSize.max,
      children: <Widget>[
        Row(
          children: <Widget>[
            Text('Row 1'),
            Padding(padding: EdgeInsets.all(16.0),),
            Text('Row 2'),
            Padding(padding: EdgeInsets.all(16.0),),
            Text('Row 3'),
          ],
        ),
      ],
    );
  }
}
```

4. Crea la clase de widget ColumnAndRowNestingWidget() después de la clase de widget RowWidget().

```
class ColumnAndRowNestingWidget extends StatelessWidget {
  const ColumnAndRowNestingWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      mainAxisSize: MainAxisSize.max,
      children: <Widget>[
        Text('Columns and Row Nesting 1',),
        Text('Columns and Row Nesting 2',),
        Text('Columns and Row Nesting 3',),
        Padding(padding: EdgeInsets.all(16.0),),
        Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: <Widget>[
            Text('Row Nesting 1'),
            Text('Row Nesting 2'),
            Text('Row Nesting 3'),
          ],
        ),
      ],
    );
  }
}
```

La columna y la fila son widgets útiles para colocarlos vertical u horizontalmente. Anidar los widgets Columna y Fila crea diseños flexibles necesarios para cada circunstancia. La anidación de widgets es fundamental para diseñar diseños de interfaz de usuario de Flutter.

Botones (Buttons)

Hay una variedad de botones para elegir, según la situación, como FloatingActionButton, FlatButton, IconButton, RaisedButton, PopupMenuButton y ButtonBar.

FloatingActionButton

El widget FloatingActionButton generalmente se coloca en la parte inferior derecha o en el centro de la pantalla principal en la propiedad Scaffold floatingActionButton. Utiliza el widget FloatingActionButtonLocation para acoplar (notch) o flotar sobre la barra de navegación. Para acoplar un botón a la barra de navegación, usa el widget BottomAppBar.

De forma predeterminada, es un botón circular, pero se puede personalizar con la forma de un estadio mediante el constructor denominado FloatingActionButton.extended. En el código de ejemplo, comentamos el botón de forma del estadio para que lo pruebes.

```
floatingActionButtonLocation: FloatingActionButtonLocation.endDocked,
floatingActionButton: FloatingActionButton(
  onPressed: () {},
  child: Icon(Icons.play_arrow),
  backgroundColor: Colors.lightGreen.shade100,
),
// or
// This creates a Stadium Shape FloatingActionButton
// floatingActionButton: FloatingActionButton.extended(
//   onPressed: () {},
//   icon: Icon(Icons.play_arrow),
//   label: Text('Play'),
// ),
bottomNavigationBar: BottomAppBar(
  hasNotch: true,
  color: Colors.lightGreen.shade100,
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: <Widget>[
      Icon(Icons.pause),
      Icon(Icons.stop),
      Icon(Icons.access_time),
      Padding(
        padding: EdgeInsets.all(32.0),
      ),
    ],
  ),
),
```

La figura 6.8 muestra el widget FloatingActionButton en la parte inferior derecha de la pantalla con la muesca habilitada.

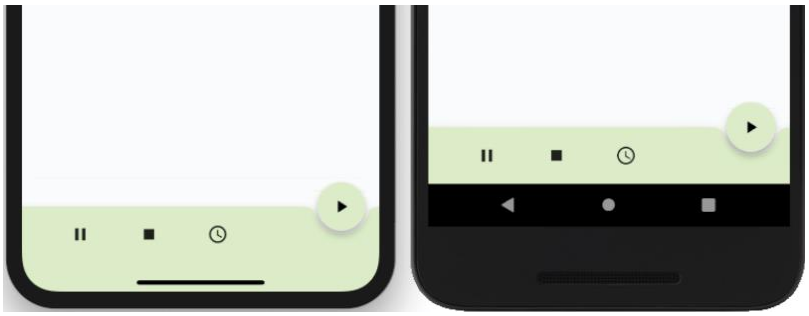


Figura 6.8: FloatingActionButton con muesca.

FlatButton

El widget FlatButton es el botón más minimalista utilizado; muestra una etiqueta de texto sin bordes ni elevación (sombra). Dado que la etiqueta de texto es un widget, puedes utilizar un widget de icono en su lugar u otro widget para personalizar el botón. color, highlightColor, splashColor, textColor y otras propiedades se pueden personalizar.

```
// Default - left button
FlatButton(
   onPressed: () {},
   child: Text('Flag'),
),

// Customize - right button
FlatButton(
   onPressed: () {},
   child: Icon(Icons.flag),
   color: Colors.lightGreen,
   textColor: Colors.white,
),
```

La figura 6.9 muestra el widget FlatButton predeterminado a la izquierda y el widget FlatButton personalizado a la derecha.

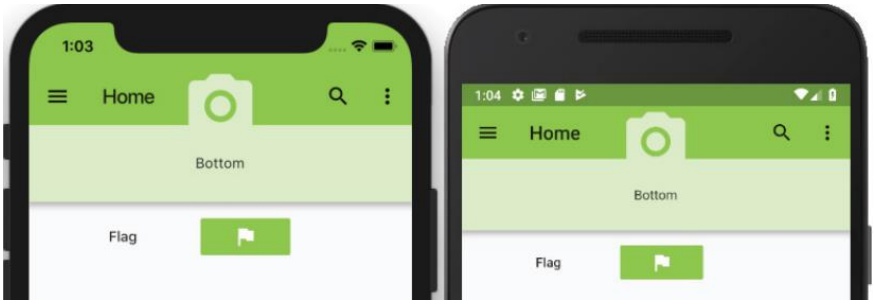


Figura 6.9: FlatButton.

RaisedButton

El widget RaisedButton agrega una dimensión y la elevación (sombra) aumenta cuando el usuario presiona el botón.

```
// Default - left button
RaisedButton(
   onPressed: () {},
   child: Text('Save'),
),
// Customize - right button
RaisedButton(
   onPressed: () {},
   child: Icon(Icons.save),
   color: Colors.lightGreen,
),
```

La figura 6.10 muestra el widget RaisedButton predeterminado a la izquierda y el widget RaisedButton personalizado a la derecha.

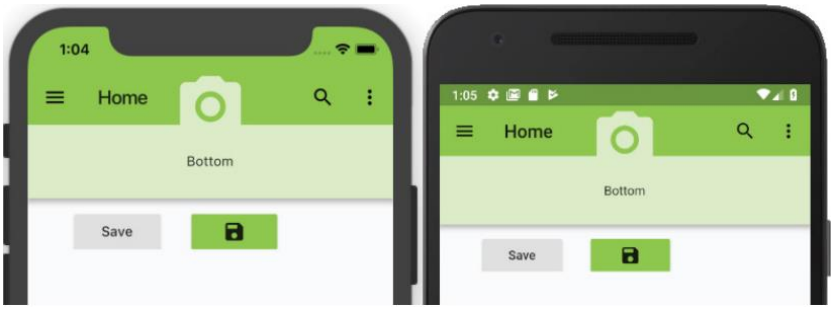


Figura 6.10: RaisedButton.

IconButton

El widget IconButton utiliza un widget de icono en un widget de componente de material que reacciona a los toques llenándose de color (tinta). La combinación crea un agradable efecto de toque, dando al usuario retroalimentación de que ha comenzado una acción.

```
// Default - left button
IconButton(
  onPressed: () {},
  icon: Icon(Icons.flight),
),
// Customize - right button
IconButton(
```

```
    onPressed: () {},
    icon: Icon(Icons.flight),
    iconSize: 42.0,
    color: Colors.white,
    tooltip: 'Flight',
  ),
```

La Figura 6.11 muestra el widget IconButton predeterminado a la izquierda y el widget IconButton personalizado a la derecha.

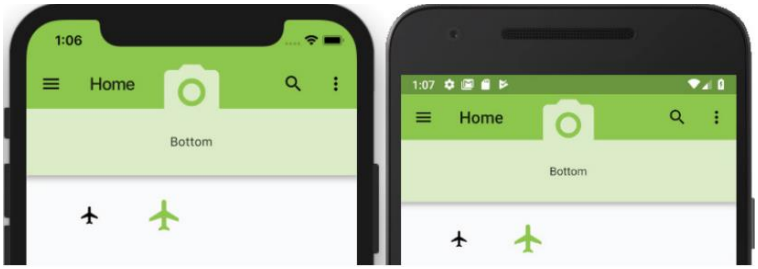


Figura 6.11: IconButton.

PopupMenuButton

El widget PopupMenuButton muestra una lista de elementos de menú. Cuando se presiona un elemento de menú, el valor pasa a la propiedad onSelected. Un uso común de este widget es colocarlo en la parte superior derecha del widget AppBar para que el usuario seleccione diferentes opciones de menú. Otro ejemplo es colocar el widget PopupMenuButton en medio del widget AppBar mostrando una lista de filtros de búsqueda.

Crear el PopupMenuButton y la clase y lista de elementos

Antes de agregar los widgets PopupMenuButton, creemos la clase y la lista necesarias para crear los elementos que se mostrarán. Por lo general, la clase TodoMenuItem(model) se crearía en un archivo Dart separado, pero para mantener el ejemplo enfocado, lo agregarás al archivo home.dart. Más adelante, separarás las clases en sus propios archivos.

1. Crea una clase TodoMenuItem. Cuando crees esta clase, asegúrate de que no esté dentro de otra clase. Crea la clase y la lista al final del archivo después del último corchete de cierre,}. La clase TodoMenuItem contiene un título y un icono.

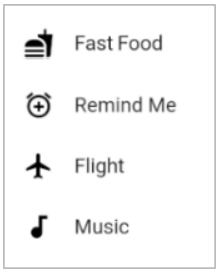
```
class TodoMenuItem {
  final String title;
  final Icon icon;
  TodoMenuItem({this.title, this.icon});
}
```

2. Crea una lista de TodoMenuItem. Esta List<TodoMenuItem> se llamará foodMenuList y contendrá una List(array) de TodoMenuItems.

```
// Create a List of Menu Item for PopupMenuButton
List<TodoMenuItem> foodMenuList = [
  TodoMenuItem(title: 'Fast Food', icon: Icon(Icons.fastfood)),
  TodoMenuItem(title: 'Remind Me', icon: Icon(Icons.add_alarm)),
  TodoMenuItem(title: 'Flight', icon: Icon(Icons.flight)),
  TodoMenuItem(title: 'Music', icon: Icon(Icons.audiotrack)),
];
```

3. Crea un PopupMenuButton. Utilizarás un itemBuilder para construir la Lista de TodoMenuItems. Si no estableces un icono para el PopupMenuButton, se utiliza un icono de menú predeterminado de forma predeterminada. El onSelected recuperará el elemento seleccionado en la lista. Utiliza itemBuilder para crear una lista de foodMenuList y mapear a TodoMenuItem. Se devuelve un PopupMenuItem para cada elemento de foodMenuList. Para el elemento secundario PopupMenuItem, usa un widget Row para mostrar los widgets Icon y Text juntos.

```
PopupMenuButton<TodoMenuItem>(
  icon: Icon(Icons.view_list),
  onSelected: ((valueSelected) {
    print('valueSelected: ${valueSelected.title}');
  }),
  itemBuilder: (BuildContext context) {
    return foodMenuList.map((TodoMenuItem todoMenuItem) {
      return PopupMenuItem<TodoMenuItem>(
        value: todoMenuItem,
        child: Row(
          children: <Widget>[
            Icon(todoMenuItem.icon.icon),
            Padding(padding: EdgeInsets.all(8.0),),
            Text(todoMenuItem.title),
          ],
        ),
      );
    }).toList();
  },
),
```



4. Modifica la propiedad bottom de AppBar agregando el nombre de la clase del widget: PopupMenuButtonWidget().

```
bottom: PopupMenuButtonWidget (),
```

5. Crea la clase de widget PopupMenuButtonWidget() después de la clase de widget ColumnAndRowNestingWidget(). Dado que la propiedad inferior espera un PreferredSizeWidget, utiliza la palabra clave implements PreferredSizeWidget en la declaración de clase. La clase extiende StatelessWidget e implementa PreferredSizeWidget.

Después de la construcción del widget, implemente el @override preferredSize getter; este es un paso obligatorio porque el propósito de PreferredSizeWidget es proporcionar el tamaño del widget; en este ejemplo, establecerás la propiedad de altura (height). Sin este paso, no tendríamos ningún tamaño especificado.

```
@override
// implement preferredSize
Size get preferredSize => Size.fromHeight(75.0);
```

A continuación, se muestra toda la clase de widget PopupMenuButtonWidget. Ten en cuenta que la propiedad height del widget Container usa la propiedad preferredSize.height que estableció en el PreferredSizeWidget getter.

```
class PopupMenuButtonWidget extends StatelessWidget implements PreferredSizeWidget {
  const PopupMenuButtonWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.lightGreen.shade100,
      height: preferredSize.height,
      width: double.infinity,
      child: Center(
        child: PopupMenuButton<TodoMenuItem>(
          icon: Icon(Icons.view_list),
          onSelected: ((valueSelected) {
            print('valueSelected: ${valueSelected.title}');
          }),
          itemBuilder: (BuildContext context) {
            return foodMenuList.map((TodoMenuItem todoMenuItem) {
              return PopupMenuItem<TodoMenuItem>(
                value: todoMenuItem,
                child: Row(
                  children: <Widget>[
                    Icon(todoMenuItem.icon.icon),
                    Padding(
                      padding: EdgeInsets.all(8.0),
                    ),
                    Text(todoMenuItem.title),
                  ],
                ),
              );
            }).toList();
          },
        ),
      ),
    );
  }

  @override
  // implement preferredSize
  Size get preferredSize => Size.fromHeight(75.0);
}
```

El widget PopupMenuButton es un gran widget para mostrar una lista de elementos, como opciones de menú. Para la lista de elementos, creaste una clase TodoMenuItem para contener un título y un icono. Creaste foodMenuList, que es una lista de cada TodoMenuItem. En este caso, los elementos de la lista están codificados, pero en una aplicación del mundo real, los valores se pueden leer desde un servicio web.

ButtonBar

El widget ButtonBar (figura 6.12) alinea los botones horizontalmente. En este ejemplo, el widget ButtonBar es un elemento secundario de un widget Container para darle un color de fondo.

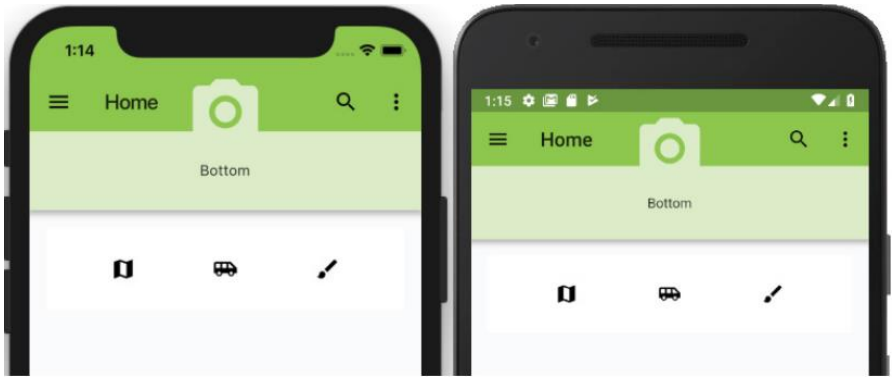


Figura 6.12: ButtonBar.

```
Container(
  color: Colors.white70,
  child: ButtonBar(
    alignment: MainAxisAlignment.spaceEvenly,
    children: <Widget>[
      IconButton(
        icon: Icon(Icons.map),
        onPressed: () {},
      ),
      IconButton(
        icon: Icon(Icons.airport_shuttle),
        onPressed: () {},
      ),
      IconButton(
        icon: Icon(Icons.brush),
        onPressed: () {},
      ),
    ],
  ),
),
```

Agregar botones como clases de widgets

Has mirado los widgets FloatingActionButton, FlatButton, RaisedButton, IconButton, PopupMenuButton y ButtonBar. Aquí crearás dos clases de widgets para organizar el diseño de los botones.

1. Agrega los nombres de clase de widget ButtonsWidget() y ButtonBarWidget() a la lista de widgets secundarios de Column. La columna se encuentra en la propiedad del cuerpo. Agrega un widget Divider() entre cada nombre de clase de widget. Asegúrate de que cada clase de widget use la palabra clave const.

```
body: Padding(
  padding: EdgeInsets.all(16.0),
  child: SafeArea(
    child: SingleChildScrollView(
      child: Column(
        children: <Widget>[
          const ContainerWithBoxDecorationWidget(),
          Divider(),
          const ColumnWidget(),
          Divider(),
          const RowWidget(),
          Divider(),
          const ColumnAndRowNestingWidget(),
          Divider(),
          const ButtonsWidget(),
          Divider(),
          const ButtonBarWidget(),
        ],
      ),
    ),
  ),
),
```

2. Crea la clase de widget ButtonsWidget() después de la clase de widget ColumnAndRowNestingWidget(). La clase devuelve una columna con tres widgets de fila para la lista secundaria de Widget. Cada lista de elementos secundarios de Row de Widget contiene diferentes botones como FlatButton, RaisedButton e IconButton. El código se presenta en la página 27.

3. Crea la clase de widget ButtonBarWidget() después de la clase de widget ButtonsWidget(). La clase devuelve un contenedor con ButtonBar como hijo. La lista secundaria del widget ButtonBar contiene tres widgets IconButton. El código se detalla en la página 28.

Los widgets FloatingActionButton, FlatButton, RaisedButton, IconButton, PopupMenuButton y ButtonBar se pueden configurar estableciendo el icono de propiedades, iconSize, información sobre herramientas, color, texto y más.

```
class ButtonsWidget extends StatelessWidget {
  const ButtonsWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Row(
          children: <Widget>[
            Padding(padding: EdgeInsets.all(16.0)),
            FlatButton(
              onPressed: () {},
              child: Text('Flag'),
            ),
            Padding(padding: EdgeInsets.all(16.0)),
            FlatButton(
              onPressed: () {},
              child: Icon(Icons.flag),
              color: Colors.lightGreen,
              textColor: Colors.white,
            ),
          ],
        ),
        Divider(),
        Row(
          children: <Widget>[
            Padding(padding: EdgeInsets.all(16.0)),
            RaisedButton(
              onPressed: () {},
              child: Text('Save'),
            ),
            Padding(padding: EdgeInsets.all(16.0)),
            RaisedButton(
              onPressed: () {},
              child: Icon(Icons.save),
              color: Colors.lightGreen,
            ),
          ],
        ),
        Divider(),
        Row(
          children: <Widget>[
            Padding(padding: EdgeInsets.all(16.0)),
            IconButton(
              icon: Icon(Icons.flight),
              onPressed: () {},
            ),
            Padding(padding: EdgeInsets.all(16.0)),
            IconButton(
              icon: Icon(Icons.flight),
              iconSize: 42.0,
              color: Colors.lightGreen,
              tooltip: 'Flight',
              onPressed: () {},
            ),
          ],
        ),
        Divider(),
      ],
    );
  }
}
```

```
class ButtonBarWidget extends StatelessWidget {
  const ButtonBarWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.white70,
      child: ButtonBar(
        alignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          IconButton(
            icon: Icon(Icons.map),
            onPressed: () {},
          ),
          IconButton(
            icon: Icon(Icons.airport_shuttle),
            onPressed: () {},
          ),
          IconButton(
            icon: Icon(Icons.brush),
            highlightColor: Colors.purple,
            onPressed: () {},
          ),
        ],
      ),
    );
  }
}
```

Usar imágenes e íconos

Las imágenes pueden hacer que una aplicación se vea tremenda o fea dependiendo de la calidad de la obra de arte. Las imágenes, los iconos y otros recursos suelen estar integrados en una aplicación.

AssetBundle

La clase AssetBundle brinda acceso a recursos personalizados como imágenes, fuentes, audio, archivos de datos y más. Antes de que una aplicación Flutter pueda usar un recurso, debes declararlo en el archivo pubspec.yaml.

```
// pubspec.yaml file to edit
# To add assets to your application, add an assets section, like this:
assets:
 —assets/images/logo.png
 —assets/images/work.png
 —assets/data/seed.json
```

En lugar de declarar cada activo (asset), que puede ser muy extenso, puedes declarar todos los activos en cada directorio. Asegúrate de terminar el nombre del directorio con una barra inclinada, /. A lo largo del curso, usaremos este enfoque cuando agreguemos activos a los proyectos.

```
// pubspec.yaml file to edit
# To add assets to your application, add an assets section, like this:
assets:
 —assets/images/
 —assets/data/
```

Imagen

El widget de imagen muestra una imagen de una fuente local o URL (web). Para cargar un widget de imagen, hay algunos constructores diferentes para usar.

- Image(): recupera la imagen de una clase ImageProvider.
- Image.asset(): recupera la imagen de una clase AssetBundle usando una llave.
- Image.file(): recupera la imagen de una clase de archivo.

- Image.memory(): recupera la imagen de una clase Uint8List.

- Image.network(): recupera la imagen de una ruta URL.

Presione Ctrl + Barra espaciadora para invocar la finalización del código para las opciones disponibles (figura 6.13).

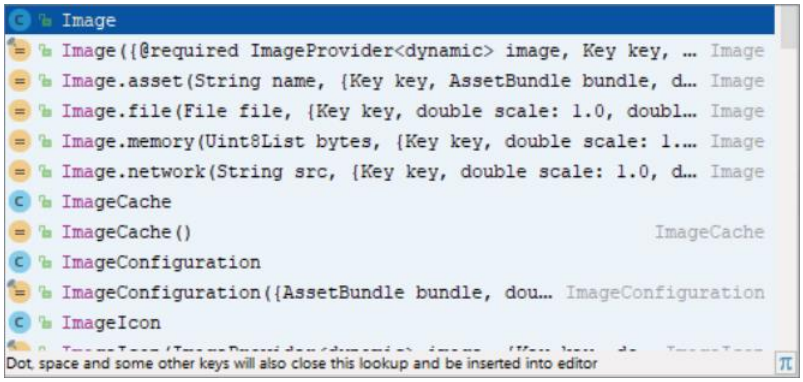


Figura 6.13: Finalización del código de la imagen.

Como nota al margen, el widget de imagen también admite GIF animados.

El siguiente ejemplo usa el constructor Image predeterminado para inicializar la imagen y ajustar argumentos. El argumento de la imagen se establece mediante el constructor AssetImage() con la ubicación predeterminada del paquete del archivo logo.png. Puedes usar el argumento de ajuste para cambiar el tamaño del widget de imagen con las opciones de BoxFit, como contain, cover, fill, fitHeight, fitWidth o none (figura 6.14).

```
// Image - on the left side
Image(
   image: AssetImage("assets/images/logo.png"),
   fit: BoxFit.cover,
),
// Image from a URL - on the right side
Image.network(
 'https://flutter.io/images/catalog-widget-placeholder.png',
),
```

Si agregas color a la imagen, se colorea la parte de la imagen y deja las transparencias solo, dando una apariencia de silueta (figura 6.15).

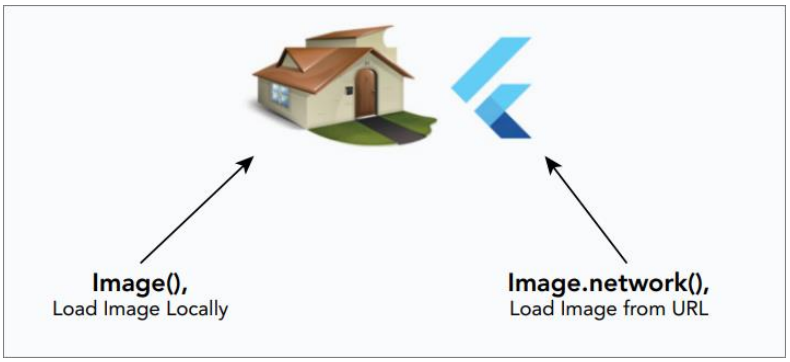


Figura 6.14: Imágenes cargadas localmente y desde la red (web).

Figura 6.15: Imagen de estilo silueta.

Icono (Icon)

El widget Icon se dibuja con un glifo de una fuente descrita en IconData. El archivo icons.dart de Flutter tiene la lista completa de iconos disponibles en la fuente MaterialIcons. Una excelente manera de agregar íconos personalizados es agregar a las fuentes AssetBundle que contienen glifos. Un ejemplo es Font Awesome, que tiene una lista de iconos de alta calidad y un paquete Flutter. Por supuesto, hay muchos otros íconos de alta calidad disponibles de otras fuentes.

El widget de icono te permite cambiar el color, el tamaño y otras propiedades del widget de icono (figura 6.16).

```
Icon(
  Icons.brush,
  color: Colors.lightBlue,
  size: 48.0,
),
```

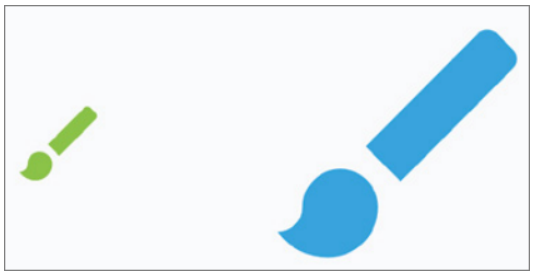


Figura 6.16: Iconos con tamaños personalizados.

Creación del proyecto de imágenes; Adición de activos; y carga de imágenes, iconos y decoradores

Crea un nuevo proyecto de Flutter y llámalo images; puedes seguir las instrucciones del tema 4. Para este proyecto, solo necesitas crear las páginas (pages) y las carpetas de assets/images. Crea la clase Home como StatelessWidget. El objetivo de esta aplicación es proporcionar una idea de cómo utilizar los widgets de imágenes e iconos.

En este ejemplo, personalizarás la propiedad de ancho (width) de los dos widgets de imagen según el tamaño de la pantalla del dispositivo. Para obtener el tamaño de la pantalla del dispositivo, puedes utilizar el método MediaQuery.of().

1. Abre el archivo pubspec.yaml para agregar recursos. En la sección de activos, agrega la declaración de la carpeta assets/images/. Nos gusta crear una carpeta de activos en la raíz del proyecto y agregar subcarpetas para cada tipo de recurso, como se muestra en el tema 4.

   ```
   # To add assets to your application, add an assets section, such as this:
   assets:
   - assets/images/
   ```

   Agrega los recursos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copia el archivo logo.png en la carpeta de imágenes. Haz clic en el botón Guardar y, según el editor que estés utilizando, ejecutarás automáticamente los paquetes de flutter get. Una vez terminado, muestra este mensaje: *Process finished with exit code 0*. Si no se ejecuta automáticamente el comando por ti, abre la ventana Terminal (ubicada en la parte inferior de tu editor) y escribe flutter packages get.

2. Abre el archivo home.dart y modifica la propiedad del cuerpo. Agrega un widget SafeArea a la propiedad del cuerpo con SingleChildScrollView como elemento secundario del widget SafeArea. Agrega Padding como elemento secundario de SingleChildScrollView y luego agrega una columna como elemento secundario del Padding.

   ```
   body: SafeArea(
     child: SingleChildScrollView(
       child: Padding(
         padding: EdgeInsets.all(16.0),
         child: Column(
           children: <Widget>[
           ],
         ),
       ),
     ),
   ),
   ```

3. Agrega el nombre de la clase de widget ImagesAndIconWidget() a la lista de widgets secundarios de Column. La columna se encuentra en la propiedad del cuerpo.

```
body: SafeArea(
  child: SingleChildScrollView(
    child: Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          const ImagesAndIconWidget(),
        ],
      ),
    ),
  ),
),
```

4. Agrega la clase de widget ImagesAndIconWidget() después de que la clase Home extienda StatelessWidget{...}. En la clase de widget, la clase AssetImage carga una imagen local. Usando el constructor Image.network, una imagen se carga mediante una cadena de URL. La propiedad de ancho del widget de imagen usa MediaQuery.of(context).size.width / 3 para calcular el valor de ancho como un tercio del ancho del dispositivo.

```
class ImagesAndIconWidget extends StatelessWidget {
  const ImagesAndIconWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: <Widget>[
        Image(
          image: AssetImage("assets/images/logo.png"),
          //color: Colors.orange,
          fit: BoxFit.cover,
          width: MediaQuery.of(context).size.width / 3,
        ),
        Image.network(
          'https://flutter.io/images/catalog-widget-placeholder.png',
          width: MediaQuery.of(context).size.width / 3,
        ),
        Icon(
          Icons.brush,
          color: Colors.lightBlue,
          size: 48.0,
        ),
      ],
    );
  }
}
```

Al declarar tus assets en el archivo pubspec.yaml, la clase AssetImage puede acceder a ellos desde un AssetBundle. El widget Image a través de la propiedad image carga una imagen local con la clase AssetBundle. Para cargar una imagen a través de la red (como la Web), usa el constructor Image.network pasando una cadena de URL. El widget Icon utiliza la biblioteca de fuentes MaterialIcons, que dibuja un glifo de la fuente descrita en la clase IconData.

Usando decoradores

Los decoradores ayudan a transmitir un mensaje según la acción del usuario o personalizan la apariencia de un widget. Hay diferentes tipos de decoradores para cada tarea.

- *Decoration*: la clase base para definir otras decoraciones.
- *BoxDecoration*: proporciona muchas formas de dibujar un cuadro con border, body y boxShadow.
- *InputDecoration*: se utiliza en TextField y TextFormField para personalizar el borde, la etiqueta, el icono y los estilos. Esta es una excelente manera de brindar comentarios al usuario sobre la entrada de datos, especificando una pista, un error, un ícono de alerta y más.

Una clase BoxDecoration (figura 6.17) es una excelente manera de personalizar un widget Container para crear formas estableciendo las propiedades borderRadius, color, gradient y boxShadow.

```
// BoxDecoration
Container(
  height: 100.0,
  width: 100.0,
  decoration: BoxDecoration(
    borderRadius: BorderRadius.all(Radius.
circular(20.0)),
    color: Colors.orange,
    boxShadow: [
      BoxShadow(
        color: Colors.grey,
        blurRadius: 10.0,
        offset: Offset(0.0, 10.0),
      )
    ],
  ),
),
```

Figura 6.17: BoxDecoration aplicado a un contenedor.

La clase InputDecoration (figura 6.18) se usa con TextField o TextFormField para especificar etiquetas, bordes, íconos, sugerencias, errores y estilos. Esto es útil para comunicarse con el usuario mientras ingresa datos. Para la propiedad de borde que se muestra aquí, estamos implementando dos formas de personalizarla, con UnderlineInputBorder y con OutlineInputBorder:

```
// TextField
TextField(
  keyboardType: TextInputType.text,
  style: TextStyle(
    color: Colors.grey.shade800,
    fontSize: 16.0,
  ),
  decoration: InputDecoration(
    labelText: "Notes",
    labelStyle: TextStyle(color: Colors.purple),
    //border: UnderlineInputBorder(),
    border: OutlineInputBorder(),

  ),
),

// TextFormField
TextFormField(
  decoration: InputDecoration(
    labelText: 'Enter your notes',
  ),
),
```

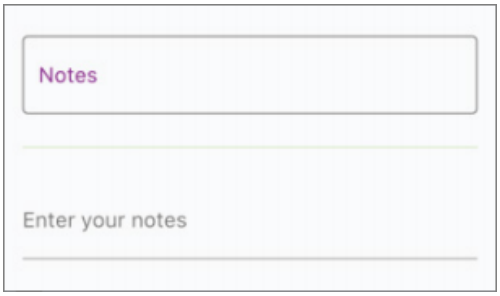


Figura 6.18: InputDecoration con OutlineInputBorder y borde predeterminado.

Continuación del proyecto de imágenes agregando decoradores

Seguiremos aun editando el archivo home.dart, agregarás las clases de widget BoxDecoratorWidget() y InputDecoratorsWidget().

1. Agrega los nombres de clase de widget BoxDecoratorWidget() y InputDecoratorsWidget() después de la clase de widget ImagesAndIconWidget(). Agrega un widget Divider() entre cada nombre de clase de widget.

```
body: SafeArea(
  child: SingleChildScrollView(
    child: Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          const ImagesAndIconWidget(),
          Divider(),
          const BoxDecoratorWidget(),
          Divider(),
          const InputDecoratorsWidget(),
        ],
      ),
    ),
  ),
),
```

2. Agrega la clase de widget BoxDecoratorWidget() después de la clase de widget ImagesAndIconWidget(). La clase de widget devuelve un widget Padding con el widget Container como hijo. La propiedad de decoración Container usa la clase BoxDecoration. Usando las propiedades de BoxDecoration borderRadius, color y boxShadow, crea una forma de botón redondeada como la de la figura 6.17.

```
class BoxDecoratorWidget extends StatelessWidget {
  const BoxDecoratorWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: EdgeInsets.all(16.0),
      child: Container(
        height: 100.0,
        width: 100.0,
        decoration: BoxDecoration(
          borderRadius: BorderRadius.all(Radius.circular(20.0)),
          color: Colors.orange,
          boxShadow: [
            BoxShadow(
              color: Colors.grey,
              blurRadius: 10.0,
              offset: Offset(0.0, 10.0),
            )
          ],
        ),
      ),
    );
  }
}
```

3. Agrega la clase de widget InputDecoratorsWidget() después de la clase de widget BoxDecoratorWidget(). Toma un TextField y usa TextStyle para cambiar las propiedades de color y fontSize. La clase InputDecoration se usa para establecer los valores labelText, labelStyle, border y enabledBorder para personalizar las propiedades del borde. Estamos usando OutlineInputBorder aquí, pero también podrías usar la clase UnderlineInputBorder en su lugar. Dejamos el borde UnderlineInputBorder y enabledBorder OutlineInputBorder() comentado, lo que te permite probar ambas clases.

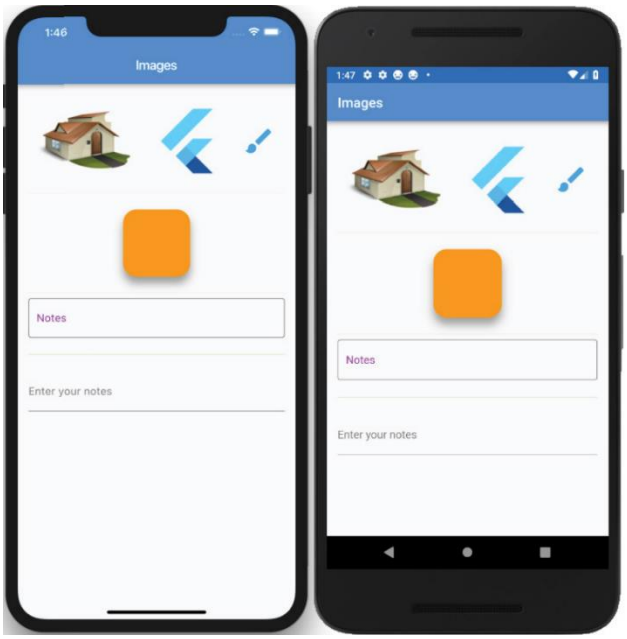
El siguiente código agrega dos widgets TextField personalizados por dos decoraciones diferentes. El primer TextField personaliza diferentes propiedades InputDecoration para mostrar una etiqueta de notas púrpura con OutlineInputBorder(). El segundo widget TextField usa la decoración sin personalizar la propiedad del borde.

```
class InputDecoratorsWidget extends StatelessWidget {
  const InputDecoratorsWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        TextField(
          keyboardType: TextInputType.text,
          style: TextStyle(
            color: Colors.grey.shade800,
            fontSize: 16.0,
          ),
          decoration: InputDecoration(
           labelText: "Notes",
           labelStyle: TextStyle(color: Colors.purple),
           //border: UnderlineInputBorder(),
           //enabledBorder: OutlineInputBorder(borderSide: BorderSide(color.
Colors.purple)),
           border: OutlineInputBorder(),
          ),
        ),
        Divider(
          color: Colors.lightGreen,
          height: 50.0,
        ),
        TextFormField(
          decoration: InputDecoration(labelText: 'Enter your notes'),
        ),
      ],
    );
  }
}
```

Los decoradores son invaluables para mejorar la apariencia de los widgets. BoxDecoration proporciona muchas formas de dibujar un cuadro con border, body y boxShadow. InputDecoration se utiliza en TextField o TextFormField. No solo permite la personalización

del borde, la etiqueta, el ícono y los estilos, sino que también brinda a los usuarios comentarios sobre la entrada de datos con sugerencias, errores, íconos y más.



Usar el widget de formulario para validar campos de texto

Hay diferentes formas de utilizar widgets de campo de texto para recuperar, validar y manipular datos. El widget de formulario (Form) es opcional, pero los beneficios de usar un widget de formulario son validar cada campo de texto como un grupo. Puedes agrupar los widgets de TextFormField para validarlos manual o automáticamente. El widget TextFormField envuelve un widget TextField para proporcionar validación cuando está incluido en un widget Form.

Si todos los campos de texto pasan el método de validación FormState, este devuelve verdadero. Si algún campo de texto contiene errores, muestra el mensaje de error correspondiente para cada campo de texto y el método de validación FormState devuelve falso. Este proceso te brinda la capacidad de usar FormState para verificar errores de validación en lugar de verificar cada campo de texto en busca de errores y no permitir la publicación de datos no válidos.

El widget Form necesita una llave única para identificarlo y se crea mediante GlobalKey. Este valor de GlobalKey es único en toda la aplicación.

En el siguiente ejemplo, crearás un formulario con dos TextFormFields (figura 6.19) para ingresar un artículo y la cantidad a ordenar. Crearás una clase de pedido para guardar el artículo y la cantidad y completar el pedido una vez que pase la validación.

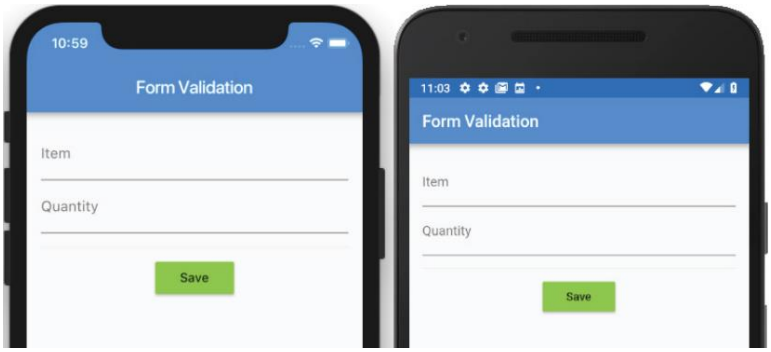


Figura 6.19: El diseño Form y TextFormField.

Creación de la aplicación de validación de formularios

Crea un nuevo proyecto Flutter y asígnale el nombre form_validation. Puedes seguir las instrucciones del tema 4. Para este proyecto, solo necesitas crear la carpeta de páginas (pages). El objetivo de esta aplicación es mostrar cómo validar los valores de entrada de datos.

1. Abre el archivo home.dart y agregue al cuerpo un widget SafeArea con Column como hijo. En los hijos de la columna, agrega el widget Form(), que modificas en el paso 7.

```
body: SafeArea(
   child: Column(
      children: <Widget>[
         Form(),
      ],
   ),
),
```

2. Crea la clase Order después de que la clase _HomeState extienda State <Home> {...}. La clase Order contendrá el artículo como valor de cadena y la cantidad como valor int.

```
class _HomeState extends State<Home> {
 //...
}
class Order {
  String item;
  int quantity;
}
```

3. Después de que la clase _HomeState extienda la declaración State<Home> y antes de @override, agrega las variables _formStateKey para el valor GlobalKey y _order para iniciar la clase Order.
   Crea la llave única para el formulario utilizando GlobalKey <FormState> y márcala como final, ya que no cambiará.

```
class _HomeState extends State<Home> {
  final GlobalKey<FormState> _formStateKey = GlobalKey<FormState>();
  // Order to Save
  Order _order = Order();
```

4. Crea el método _validateItemRequired (String value) que acepta un valor de cadena. Utiliza el operador ternario para comprobar si el valor está establecido en isEmpty y, en caso afirmativo, devuelva 'Item Required'. De lo contrario, devuelve null.

```
String _validateItemRequired(String value) {
  return value.isEmpty ? 'Item Required' : null;
}
```

5. Crea el método _validateItemCount (String value) que acepta un valor de cadena. Utiliza el operador ternario para convertir String en int. Luego verifica si int es mayor que cero; si no es así, devuelve 'At least one Item is Required'.

```
String _validateItemCount(String value) {
  // Check if value is not null and convert to integer
  int _valueAsInteger = value.isEmpty ? 0 : int.tryParse(value);
  return _valueAsInteger == 0 ? 'At least one Item is Required' : null;
}
```

6. Crea el método _submitOrder() llamado por el widget FlatButton para comprobar si todos los campos de TextFormField pasan la validación y llaman a Form save() para recopilar valores de todos los TextFormFields a la clase Order.

```
void _submitOrder() {
  if(_formStateKey.currentState.validate()) {
    _formStateKey.currentState.save();
    print('Order Item: ${order.item}');
    print('Order Quantity: ${order.quantity}');
  }
}
```

7. Agrega al widget Form() una variable de llave privada llamada _formStateKey, establece autovalidate en verdadero, agrega Padding para la propiedad secundaria y agrega Column como elemento secundario de Padding.
   Establecer autovalidate en verdadero permite que el widget Form() verifique la validación de todos los campos cuando el usuario ingrese información y muestre un

mensaje apropiado. Si autovalidate se establece en false, no se realiza ninguna validación hasta que se llama manualmente al método _formStateKey.currentState.validate().

```
Form(
  key: _formStateKey,
  autovalidate: true,
  child: Padding(
    padding: EdgeInsets.all(16.0),
    child: Column(
      children: <Widget>[
      ],
    ),
  ),
),
```

8. Agrega dos widgets TextFormField a la lista Column child de Widget. El primer TextFormField es una descripción del artículo y el segundo TextFormField es una cantidad de artículos para ordenar.

9. Agrega una clase InputDecoration con hintText y labelText para cada TextFormField.

```
hintText: 'Espresso',
labelText: 'Item',
```

10. Agrega una llamada para los métodos validator y onSaved. El método validator se llama para validar los caracteres a medida que se ingresan, y el método de onSaved se llama por el método Form save() para recopilar valores de cada TextFormField.
    Para el validator, pasa el valor ingresado en el widget TextFormField nombrando el valor de la variable entre paréntesis y usa la sintaxis de flecha gruesa (=>) para llamar al método _validateItemRequired (value). La sintaxis de la flecha gruesa es una abreviatura de {return mycustomexpression; }.

```
validator: (value) => _validateItemRequired(value),
```

Ten en cuenta que en el paso 2 creaste una clase Order para contener los valores de artículo y cantidad que se recopilarán mediante los métodos onSaved. Cuando se llama al método Form save(), se llaman a todos los métodos TextFieldForm onSaved y los valores se recopilan en la clase Order, como order.item = value.

```
onSaved: (value) => order.item = value,
```

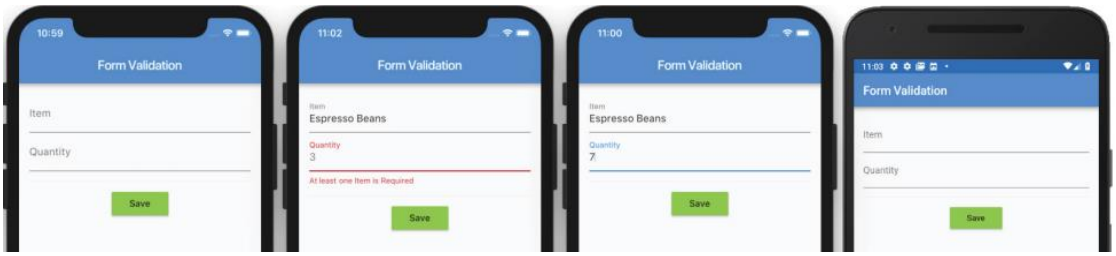
El siguiente código muestra ambos TextFormFields:

```
TextFormField(
  decoration: InputDecoration(
    hintText: 'Espresso',
    labelText: 'Item',
  ),
  validator: (value) => _validateItemRequired(value),
  onSaved: (value) => order.item = value,
),
TextFormField(
  decoration:  InputDecoration(
    hintText: '3',
    labelText: 'Quantity',
  ),
  validator: (value) => _validateItemCount(value),
  onSaved: (value) => order.quantity = int.tryParse(value),
),
```

Observa que usa int.tryParse() para convertir el valor de la cantidad de String a int.

11. Agrega un Divider y RaisedButton después del último TextFormField. Para onPressed, llama al método _submitOrder() creado en el paso 6.

```
Divider(height: 32.0,),
RaisedButton(
    child: Text('Save'),
    color: Colors.lightGreen,
    onPressed: () => _submitOrder(),
),
```



Al recuperar datos de los campos de entrada, el widget Form es una ayuda increíble, y usaste la clase GlobalKey para asignar una llave única para identificarlo. Utiliza el widget de formulario para agrupar los widgets de TextFormField para validarlos manual o automáticamente. El método de validación FormState valida los datos y, si pasa, devuelve verdadero. Si el método de validación FormState falla, devuelve falso y cada campo de texto muestra el mensaje de error correspondiente.

Cada propiedad de validación de TextFormField tiene un método para verificar el valor apropiado. Cada propiedad onSaved de TextFormField pasa el valor ingresado actualmente a la clase Order. En una aplicación del mundo real, tomaría los valores de la clase Order y los guardaría en una base de datos localmente o en un servidor web.

Comprobar la orientación

En ciertos escenarios, conocer la orientación del dispositivo ayuda a diseñar la interfaz de usuario adecuada. Hay dos formas de averiguar la orientación, MediaQuery.of(context) .orientación y OrientationBuilder.

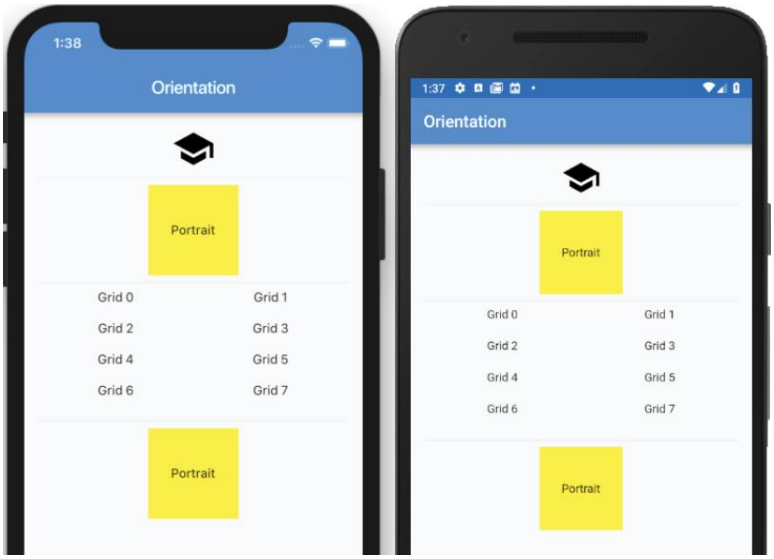
Una gran nota sobre OrientationBuilder: devuelve la cantidad de espacio disponible para que el padre descubra la orientación. Esto significa que no garantiza la orientación real del dispositivo. Se prefiere usar MediaQuery para obtener la orientación real del dispositivo debido a su precisión.

Creación de la aplicación de orientación

Crea un nuevo proyecto de Flutter y asígnale el nombre orientation. Puedes seguir las instrucciones del tema 4. Para este proyecto, solo necesitas crear la carpeta de pages.

En este ejemplo, el diseño de la interfaz de usuario cambiará según la orientación. Cuando el dispositivo está en modo vertical, mostrará un ícono, y cuando esté en modo horizontal, mostrará dos íconos. Verás un widget de contenedor que aumentará de tamaño y cambiará de color, y usará un widget de GridView para mostrar dos o cuatro columnas.

Por último, agrega el widget OrientationBuilder para mostrar que cuando OrientationBuilder no es un widget principal, la orientación correcta no se calcula correctamente. Pero si colocas el OrientationBuilder como padre, funciona correctamente; ten en cuenta que el uso de SafeArea no afecta el resultado. La siguiente imagen muestra el proyecto final.



1. Abre el archivo home.dart y agrega al cuerpo un SafeArea con SingleChildScrollView cuando era child. Agrega Padding como elemento secundario de SingleChildScrollView. Agrega una columna como elemento secundario del relleno. En la propiedad Column children, agrega la clase de widget denominada OrientationLayoutIconsWidget(), que crearás a continuación. Asegúrate de agregar la palabra clave const antes del nombre de la clase del widget para aprovechar el almacenamiento en caché y mejorar el rendimiento.

```
body: SafeArea(
  child: SingleChildScrollView(
    child: Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          const OrientationLayoutIconsWidget(),
        ],
      ),
    ),
  ),
),
```

2. Agrega la clase de widget OrientationLayoutIconsWidget() después de que la clase Home extienda StatelessWidget{...}. La primera variable para inicializar es la orientación actual llamando a MediaQuery.of () después de la construcción del widget (BuildContext context).

```
class OrientationLayoutIconsWidget extends StatelessWidget {
  const OrientationLayoutIconsWidget({
    Key key,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
    Orientation _orientation = MediaQuery.of(context).orientation;
    return Container();
  }
}
```

3. Según la Orientation actual, devuelve un diseño diferente de los widgets de iconos. Utiliza un operador ternario para comprobar si la Orientation es portrait y, de ser así, devuelva un solo icono de Fila. Si la Orientation es landscape, devuelve una Fila de dos widgets de iconos. Reemplaza el actual return Container() con el siguiente código:

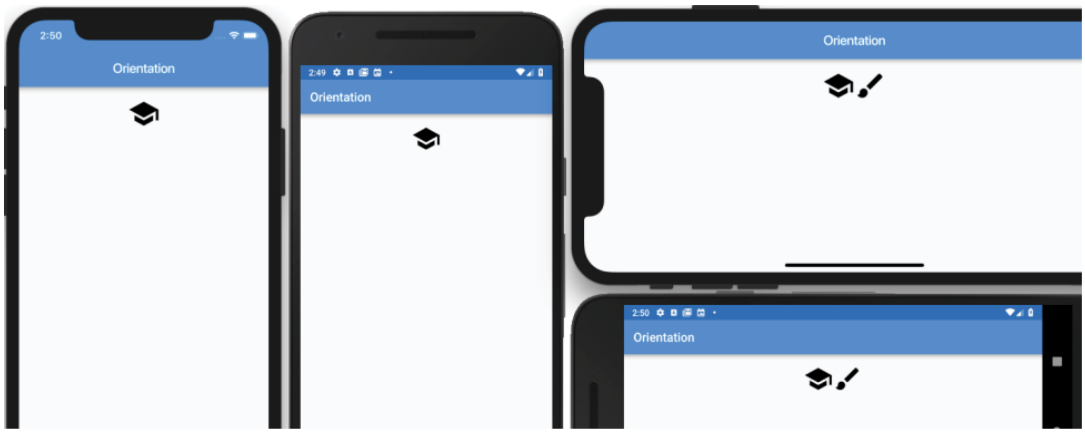
```
return _orientation == Orientation.portrait
    ? Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Icon(
          Icons.school,
          size: 48.0,
        ),
      ],
    )
    : Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Icon(
          Icons.school,
          size: 48.0,
        ),
        Icon(
          Icons.brush,
          size: 48.0,
        ),
      ],
);
```

4. Juntando todo el código, se obtiene lo siguiente:

```
class OrientationLayoutIconsWidget extends StatelessWidget {
  const OrientationLayoutIconsWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    Orientation _orientation = MediaQuery.of(context).orientation;
    return _orientation == Orientation.portrait
        ? Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Icon(
          Icons.school,
          size: 48.0,
        ),
      ],

    )
        : Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Icon(
          Icons.school,
          size: 48.0,
        ),
        Icon(
          Icons.brush,
          size: 48.0,
        ),
      ],
    );
  }
}
```



5. Después de OrientationLayoutIconsWidget (), agrega un widget Divider y la clase de widget OrientationLayoutWidget() para crear.
   Los pasos son similares a los anteriores, pero en lugar de usar filas e íconos, estás usando contenedores: obtén el modo Orientation y para el portrait devuelve un widget Container amarillo con un ancho de 100.0 píxeles. Cuando se gira el dispositivo, el paisaje devuelve un widget de Container verde con un ancho de 200,0 píxeles.

```
class OrientationLayoutWidget extends StatelessWidget {
  const OrientationLayoutWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    Orientation _orientation = MediaQuery.of(context).orientation;

    return _orientation == Orientation.portrait
        ? Container(
      alignment: Alignment.center,
      color: Colors.yellow,
      height: 100.0,
      width: 100.0,
      child: Text('Portrait'),
    )
        : Container(
      alignment: Alignment.center,
      color: Colors.lightGreen,
      height: 100.0,
      width: 200.0,
      child: Text('Landscape'),
    );
  }
}
```

6. Después de OrientationLayoutWidget(), agrega un widget Divider y la clase de widget GridViewWidget() que crearás.

Aunque examinarás más de cerca el widget GridView más adelante, es apropiado usarlo ahora ya que es el más cercano a un ejemplo del mundo real. En modo portrait, el widget GridView muestra dos columnas y en modo landscape, muestra cuatro columnas.
Hay algunos elementos a tener en cuenta aquí. Dado que el widget GridView está dentro de un widget Column, establece el argumento shrinkWrap del constructor GridView.count en verdadero o romperás las restricciones. También configura el argumento de física en NeverScrollableScrollPhysics() o GridView desplazará tus elementos secundarios desde dentro. Recuerda, tienes todos estos widgets dentro de SingleChildScrollView.

```
class GridViewWidget extends StatelessWidget {
  const GridViewWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    Orientation _orientation = MediaQuery.of(context).orientation;

    return GridView.count(
      shrinkWrap: true,
      physics: NeverScrollableScrollPhysics(),
      crossAxisCount: _orientation == Orientation.portrait ? 2 : 4,
      childAspectRatio: 5.0,
      children: List.generate(8, (int index) {
        return Text("Grid $index", textAlign: TextAlign.center,);
      }),
    );
  }
}
```

7. Después de GridViewWidget(), agrega un widget Divider y la clase de widget OrientationBuilderWidget() que crearás.
   Como se mencionó anteriormente, utilizzamos MediaQuery.of() para obtener orientación porque es más preciso, pero es bueno saber cómo usar OrientationBuilder.

OrientationBuilder requiere que se pase una propiedad de constructor y no puede ser nula. La propiedad del constructor toma dos parámetros: BuildContext y Orientation.

builder: (BuildContext context, Orientation orientation) {}

Los pasos y el resultado son los mismos que con _buildOrientationLayout(). Utiliza el operador ternario para comprobar la orientación y el portrait, y devuelve un widget de contenedor amarillo con un ancho de 100,0 píxeles. Cuando se gira el dispositivo, el landscape devuelve un widget de contenedor verde con un ancho de 200,0 píxeles.

Ten en cuenta que OrientationBuilder corre el riesgo de no detectar el modo de orientación correctamente porque es un widget secundario y depende del tamaño de la pantalla principal en lugar de la orientación del dispositivo.
Debido a esto, se recomienda usar MediaQuery.of() en su lugar.

```
// OrientationBuilder as a child does not give correct Orientation. i.e Child
of Column...
// OrientationBuilder as a parent gives correct Orientation
class OrientationBuilderWidget extends StatelessWidget {
  const OrientationBuilderWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return OrientationBuilder(
      builder: (BuildContext context, Orientation orientation) {
        return orientation == Orientation.portrait
            ? Container(
          alignment: Alignment.center,
          color: Colors.yellow,
          height: 100.0,
          width: 100.0,
          child: Text('Portrait'),
        )
            : Container(
          alignment: Alignment.center,
          color: Colors.lightGreen,
          height: 100.0,
          width: 200.0,
          child: Text('Landscape'),
        );
      },
    );
  }
}
```

Puedes detectar la orientación del dispositivo llamando a MediaQuery.of(context).orientation, que devuelve un valor vertical (portrait) u horizontal (landscape).

También está OrientationBuilder, que devuelve la cantidad de espacio disponible para que el padre averigüe la orientación. Se recomienda usar MediaQuery para recuperar la orientación correcta del dispositivo.

Resumen

En este largo tema!, aprendiste sobre los widgets (básicos) más utilizados. Estos widgets básicos son los componentes básicos para diseñar aplicaciones móviles. También exploraste diferentes tipos de botones para elegir según la situación. Aprendiste cómo agregar activos a tu aplicación a través de AssetBundle enumerando elementos en el archivo pubspec.yaml. Usaste el widget de imagen para cargar imágenes desde el dispositivo local o un servidor web a través de una cadena de URL. Viste cómo el widget Icon te da la capacidad de cargar íconos usando la biblioteca de fuentes MaterialIcons.

Para modificar la apariencia de los widgets, aprendiste a usar BoxDecoration. Para mejorar la retroalimentación de los usuarios sobre la entrada de datos, implementaste InputDecoration. La validación de múltiples entradas de datos de campos de texto puede resultar engorrosa, pero puedes utilizar el widget Formulario para validarlas de forma manual o automática. Por último, usar MediaQuery para averiguar la orientación actual del dispositivo es extremadamente poderoso en cualquier aplicación móvil para diseñar widgets según la orientación.

En el próximo tema, aprenderás a usar animaciones. Comenzarás usando widgets como AnimatedContainer, AnimatedCrossFade y AnimatedOpacity y terminarás con el poderoso AnimationController para la animación personalizada.