6

Using Common Widgets

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to use basic widgets such as `Scaffold`, `AppBar`, `SafeArea`, `Container`, `Text`, `RichText`, `Column`, and `Row`, as well as different types of buttons
- How to nest the Column and Row widgets together to create different UI layouts
- Ways to include images, icons, and decorators
- How to use text field widgets to retrieve, validate, and manipulate data
- How to check your app's orientation

In this chapter, you'll learn how to use the most common widgets. I call them our base building blocks for creating beautiful UIs and UXs. You'll learn how to load images locally or over the Web via a uniform resource locator (URL), use the included rich Material Components icons, and apply decorators to enhance the look and feel of widgets or use them as input guides to entry fields. You'll also explore how to take advantage of the `Form` widget to validate text field entry widgets as a group, not just individually. Additionally, to account for the variety of device sizes, you'll see how using the `MediaQuery` or `OrientationBuilder` widget is a great way to detect orientation—because using the device orientation and layout widgets accordingly based on portrait or landscape is extremely important. For example, if the device is in portrait mode, you can show a row of three images, but when the device is turned to landscape mode, you can show a row of five images since the width is a larger area than in portrait mode.

USING BASIC WIDGETS

When building a mobile app, you'll usually implement certain widgets for the base structure. Being familiar with them is necessary.

Beginning Flutter®: A Hands On Guide To App Development, First Edition.
Marco L. Napoli.

`Scaffold` As you learned in Chapter 4, "Creating a Starter Project Template," the `Scaffold` widget implements the basic Material Design visual layout, allowing you to easily add various widgets such as `AppBar`, `BottomAppBar`, `FloatingActionButton`, `Drawer`, `SnackBar`, `BottomSheet`, and more.

`AppBar` The `AppBar` widget usually contains the standard `title`, `toolbar`, `leading`, and `actions` properties (along with buttons), as well as many customization options.

`title` The `title` property is typically implemented with a `Text` widget. You can customize it with other widgets such as a `DropdownButton` widget.

`leading` The `leading` property is displayed before the `title` property. Usually this is an `IconButton` or `BackButton`.

`actions` The `actions` property is displayed to the right of the `title` property. It's a list of widgets aligned to the upper right of an `AppBar` widget usually with an `IconButton` or `PopupMenuButton`.

`flexibleSpace` The `flexibleSpace` property is stacked behind the `Toolbar` or `TabBar` widget. The height is usually the same as the `AppBar` widget's height. A background image is commonly applied to the `flexibleSpace` property, but any widget, such as an `Icon`, could be used.

`SafeArea` The `SafeArea` widget is necessary for today's devices such as the iPhone X or Android devices with a notch (a partial cut-out obscuring the screen usually located on the top portion of the device). The `SafeArea` widget automatically adds sufficient padding to the child widget to avoid intrusions by the operating system. You can optionally pass a minimum amount of padding or a Boolean value to not enforce padding on the top, bottom, left, or right.

`Container` The `Container` widget is a commonly used widget that allows customization of its `child` widget. You can easily add properties such as `color`, `width`, `height`, `padding`, `margin`, `border`, `constraint`, `alignment`, `transform` (such as rotating or sizing the widget), and many others. The `child` property is optional, and the `Container` widget can be used as an empty placeholder (invisible) to add space between widgets.

`Text` The `Text` widget is used to display a string of characters. The `Text` constructor takes the arguments `string`, `style`, `maxLines`, `overflow`, `textAlign`, and others. A *constructor* is how the arguments are passed to initialize and customize the `Text` widget.

`RichText` The `RichText` widget is a great way to display text using multiple styles. The `RichText` widget takes `TextSpan`s as children to style different parts of the strings.

`Column` A `Column` widget displays its children vertically. It takes a `children` property containing an array of `List<Widget>`, meaning you can add multiple widgets. The children align vertically without taking up the full height of the screen. Each child widget can be embedded in an `Expanded` widget to fill the available space. `CrossAxisAlignment`, `MainAxisAlignment`, and `MainAxisSize` can be used to align and size how much space is occupied on the main axis.

`Row` A `Row` widget displays its children horizontally. It takes a `children` property containing an array of `List<Widget>`. The same properties that the `Column` contains are applied to the `Row` widget with the exception that the alignment is horizontal, not vertical.

Buttons There are a variety of buttons to choose from for different situations such as `RaisedButton`, `FloatingActionButton`, `FlatButton`, `IconButton`, `PopupMenuButton`, and `ButtonBar`.

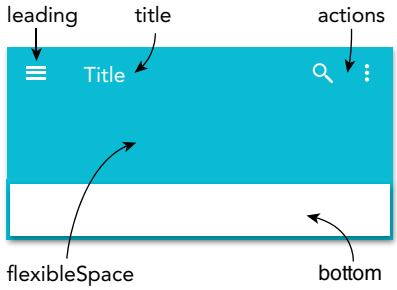
TRY IT OUT Adding AppBar Widgets

Create a new Flutter project and name it `ch6_basics`; you can follow the instructions in Chapter 4. For this project, you need to create only the `pages` folder. The goal of this app is to provide a look at how to use the basic widgets, not necessarily to design the best-looking UI. In Chapter 10, "Building Layouts," you'll focus on building complex and beautiful layouts.

1. Open the `main.dart` file. Change the `primarySwatch` property from `blue` to `lightGreen`.

```
primarySwatch: Colors.lightGreen,
```

2. Open the `home.dart` file. Start by customizing the `AppBar` widget properties.



3. Add to the `AppBar` a `leading` `IconButton`. If you override the `leading` property, it is usually an `IconButton` or `BackButton`.

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

4. The `title` property is usually a `Text` widget, but it can be customized with other widgets such as a `DropdownButton`. By following the instructions from Chapter 4, you have already added the `Text` widget to the `title` property; if not, add the `Text` widget with a value of `'Home'`.

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

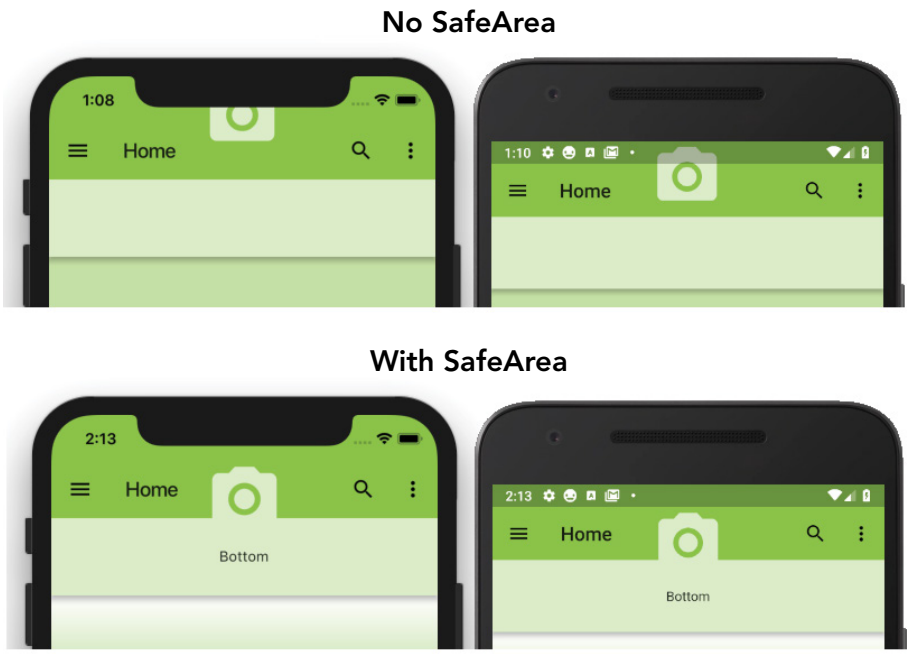
5. The `actions` property takes a list of widgets; add two `IconButton` widgets.

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

6. Because you are using an `Icon` for the `flexibleSpace` property, let's add a `SafeArea` and an `Icon` as a `child`.

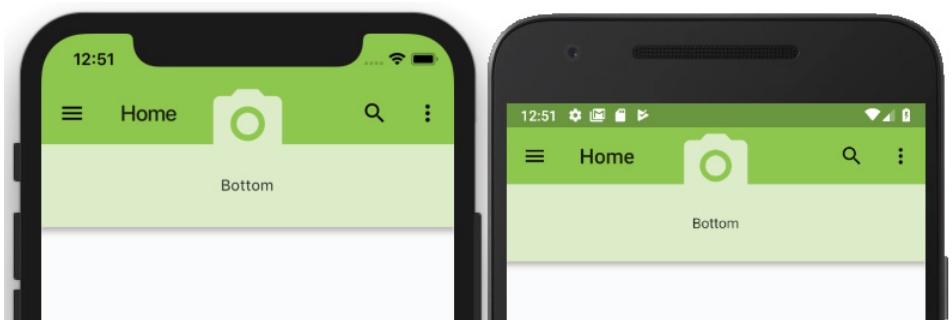
```
flexibleSpace: SafeArea(
  child: Icon(
```

```
      Icons.photo_camera,
      size: 75.0,
      color: Colors.white70,
    ),
  ),
```



7. Add a `PreferredSize` for the `bottom` property with a `Container` for a `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),
),
```



How It Works

You learned how to customize the `AppBar` widget by using widgets to set the `title`, `toolbar`, `leading`, and `actions` properties. All the properties that you learned about in this example are related to customizing the `AppBar`.

In Chapter 9, "Creating Scrolling Lists and Effects" you'll learn to use the `SliverAppBar` widget, which is an `AppBar` embedded in a `sliver` using a `CustomScrollView`, making any app come to life with pinpoint customizations such as parallax animation. I absolutely love using `slivers` because they add an extra layer of customization.

In the next section, you'll learn how to customize the `Scaffold` `body` property by nesting widgets to build the page content.

SafeArea

The `SafeArea` widget is a must for today's devices such as the iPhone X or Android devices with a notch (a partial cut-out obscuring the screen usually located on the top portion of the device). The `SafeArea` widget automatically adds sufficient padding to the child widget to avoid intrusions by the operating system. You can optionally pass minimum padding or a Boolean value to not enforce padding on the top, bottom, left, or right.

TRY IT OUT Adding a SafeArea to the Body

Continue modifying the `home.dart` file.

Add a `Padding` widget to the `body` property with a `SafeArea` as a `child`. Because this example packs in different uses of widgets, add a `SingleChildScrollView` as a `child` of the `SafeArea`. The `SingleChildScrollView` allows the user to scroll and view hidden widgets; otherwise, the user sees a yellow and black bar conveying that the widgets are overflowing.

```
body: Padding(
  padding: EdgeInsets.all(16.0),
```

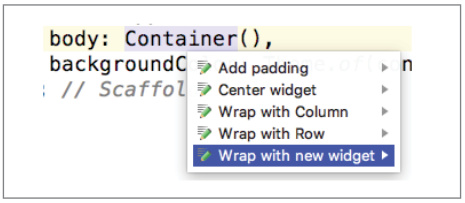
```
      child: SafeArea(
        child: SingleChildScrollView(
          child: Column(
            children: <Widget>[

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

Sample Wrap Widget Tip

There is a great way to wrap a current widget as a child of another widget. Place your cursor on top of the current widget to wrap and then press Option+Enter on your keyboard. The Dart/quick assist pops up. Choose the Wrap with new widget option.

Do not add the following steps to your project; this is a tip on how to quickly wrap a widget with another.

1. Place the cursor on the widget to wrap.
2. Press Option+Enter (Alt+Enter in Windows). The Dart/quick assist pops up.



3. Select a Wrap with new widget option such as `body: widget(child: Container()),`.
4. Rename the widget to `SafeArea` and notice that `child:` is automatically the `Container()` widget. Make sure you add a comma after the `Container()` widget, as shown here. Placing a comma after every property ensures correct Flutter formatting over multiple lines.

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

How It Works

Adding the `SafeArea` widget automatically adjusts padding for devices that have a notch. Any `SafeArea` `child` widgets are constrained to the correct padding.

Container

The `Container` widget has an optional `child` widget property and can be used as a decorated widget with a custom border, color, constraint, alignment, transform (such as rotating the widget), and more. This widget can be utilized as an empty placeholder (invisible), and if a child is omitted, it sizes to the full available screen size.

TRY IT OUT Adding a Container

Continue modifying the `home.dart` file. Since you want to keep your code readable and manageable, you'll create widget classes to build each `body` widget section of the `Column` list of widgets.

1. Add to the body property a `Padding` widget with the `child` property set to a `SafeArea` widget. Add to the `SafeArea` `child` a `SingleChildScrollView`. Add to the `SingleChildScrollView` `child` a `Column`. For the `Column` `children`, add the call to the `ContainerWithBoxDecorationWidget()` widget class, which you will create next. Make sure the widget class uses the `const` keyword to take advantage of caching (performance).

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

2. Create the `ContainerWithBoxDecorationWidget()` widget class after `class Home extends StatelessWidget {...}`. The widget class will return a `Widget`. Note that when you refactor by creating widget classes, they are of type `StatelessWidget` unless you specify to use a `StatefulWidget`.

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

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Container(),
      ],
    );
  }
}
```

3. Start adding properties to the `Container` by adding a height of 175.0 pixels. Note the comma after the number, which separates properties and helps to keep the Dart code formatted. Go to the next line to add the `decoration` property, which accepts a `BoxDecoration` class. The `BoxDecoration` class provides different ways to draw a box, and in this case, you are adding a `BorderRadius` class to the `bottomLeft` and `bottomRight` of the `Container`.

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

4. Using the named constructor `BorderRadius.only()` allows you to control the sides to draw round corners. I purposely made the `bottomLeft` radius much bigger than the `bottomRight` to show the custom shapes you can create.

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

The `BoxDecoration` also supports a `gradient` property. You are using a `LinearGradient`, but you could also have used a `RadialGradient`. The `LinearGradient` displays the gradient colors linearly, and the `RadialGradient` displays the gradient colors in a circular manner. The `begin` and `end` properties allow you to choose the start and end positions for the gradient by using the `AlignmentGeometry` class. `AlignmentGeometry` is a base class for `Alignment` that allows direction-aware resolution. You have many directions to choose from such as `Alignment.bottomLeft`, `Alignment.centerRight`, and more.

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

The `colors` property requires a `List` of `Color` types, `List<Color>`. The list of `Colors` is entered within square brackets separated by commas.

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

Here's the full gradient property source code:

```
gradient: LinearGradient(
  begin: Alignment.topCenter,
  end: Alignment.bottomCenter,
  colors: [
     Colors.white,
     Colors.lightGreen.shade500,
  ],
),
```

5. The `boxShadow` property is a great way to customize a shadow, and it takes a list of `BoxShadows`, called `List<BoxShadow>`. For the `BoxShadow`, set the `color`, `blurRadius`, and `offset` properties.

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

The last part of the `Container` is to add a child `Text` widget wrapped by a `Center` widget. The `Center` widget allows you to center the `child` widget on the screen.

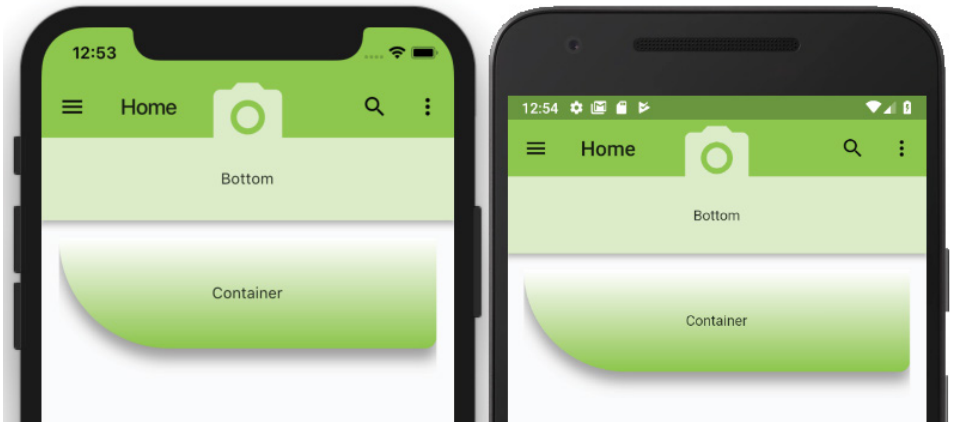
```
child: Center(
  child: Text('Container'),
),
```

6. Add a `Center` widget as a `child` of the `Container`, and add to the `Center` widget `child` a `Text` widget with the string `Container`. (In the next section, I'll go over the `Text` widget in detail.)

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

Here's the full `ContainerWithBoxDecorationWidget()` widget class source code:

```
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'),          ),
        ),
      ],
    );
  }
}
```



How It Works

Containers can be powerful widgets full of customization. By using decorators, gradients, and shadows, you can create beautiful UIs. I like to think of containers as enhancing an app in the same way a great-looking frame adds to a painting.

Text

You've already used the `Text` widget in the preceding examples; it's an easy widget to use but also customizable. The `Text` constructor takes the arguments `string`, `style`, `maxLines`, `overflow`, `textAlign`, and others.

```
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,
),
```

RichText

The `RichText` widget is a great way to display text using multiple styles. The `RichText` widget takes `TextSpan` as children to style different parts of the strings (Figure 6.1).

FIGURE 6.1: RichText with TextSpan

TRY IT OUT Replacing Text with a RichText Child Container

Instead of using the previous `Container` `Text` widget to show a plain-text property, you can use a `RichText` widget to enhance and emphasize the words in your string. You can change each word's color and styles.

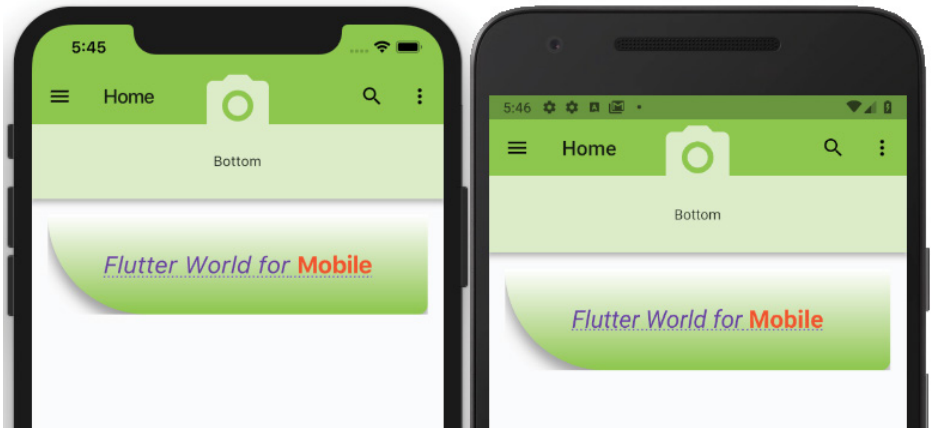
1. Find the `Container` child `Text` widget and delete `Text('Container')`.

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

2. Replace the `Container` child's `Text` widget with a `RichText` widget. The `RichText` `text` property is a `TextSpan` object (class) that is customized by using a `TextStyle` for the `style` property. The `TextSpace` has a `children` list of `TextSpan` where you place different `TextSpan` objects to format different portions of the entire `RichText`.

   By using the `RichText` widget and combining different `TextSpan` objects, you create rich-text formatting like with a word processor.

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



How It Works

`RichText` is a powerful widget when combined with the `TextSpan` object (class). There are two main parts to styling, the default `text` property and the `children` list of `TextSpan`. The `text` property using a `TextSpan` sets the default styling for the `RichText`. The `children` list of `TextSpan` allows you to use multiple `TextSpan` objects to format different strings.

Column

A `Column` widget (Figures 6.2 and 6.3) displays its children vertically. It takes a `children` property containing an array of `List<Widget>`. The children align vertically without taking up the full height of the screen. Each child widget can be embedded in an `Expanded` widget to fill available space. You can use `CrossAxisAlignment`, `MainAxisAlignment`, and `MainAxisSize` to align and size how much space is occupied on the main axis.

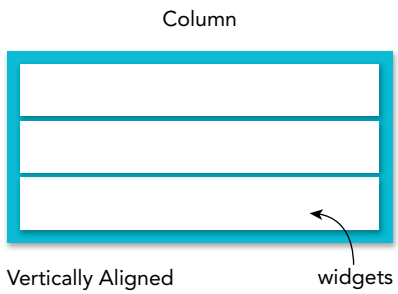


FIGURE 6.2: Column widget

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

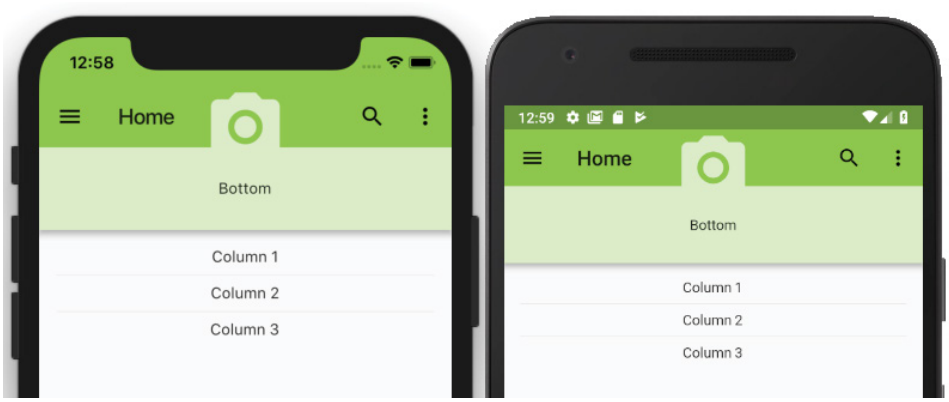


FIGURE 6.3: Column widget rendered in app

Row

A `Row` widget (Figures 6.4 and 6.5) displays its children horizontally. It takes a `children` property containing an array of `List<Widget>`. The same properties that the `Column` contains are applied to the `Row` widget with the exception that the alignment is horizontal, not vertical.

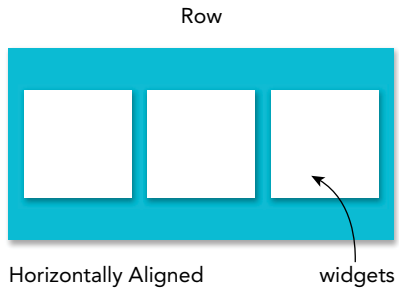


FIGURE 6.4: Row widget

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

Column and Row Nesting

A great way to create unique layouts is to combine `Column` and `Row` widgets for individual needs. Imagine having a journal page with `Text` in a `Column` with a nested `Row` containing a list of images (Figures 6.6 and 6.7).

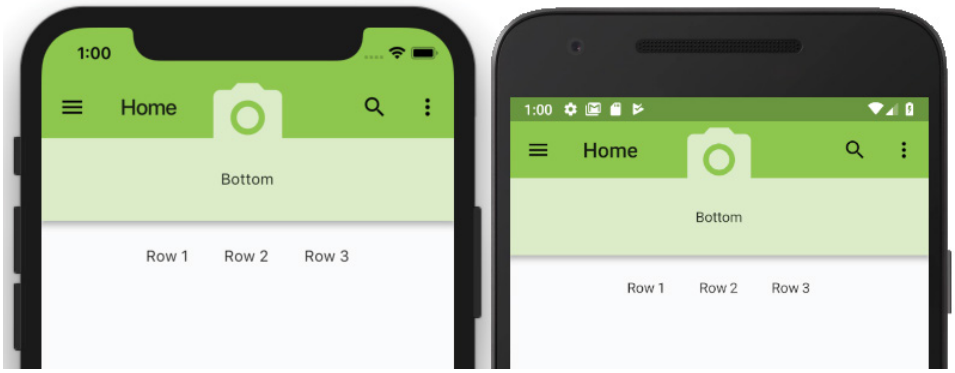


FIGURE 6.5: Row widget rendered in app

Add a `Row` widget inside the `Column` widget. Use `mainAxis-Alignment: MainAxisAlignment.spaceEvenly` and add three `Text` widgets.

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

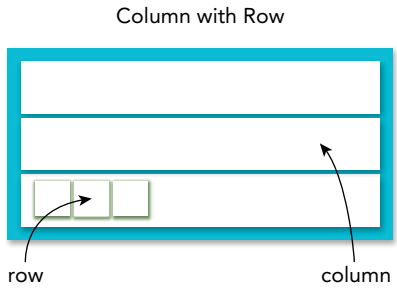


FIGURE 6.6: Column and Row nesting

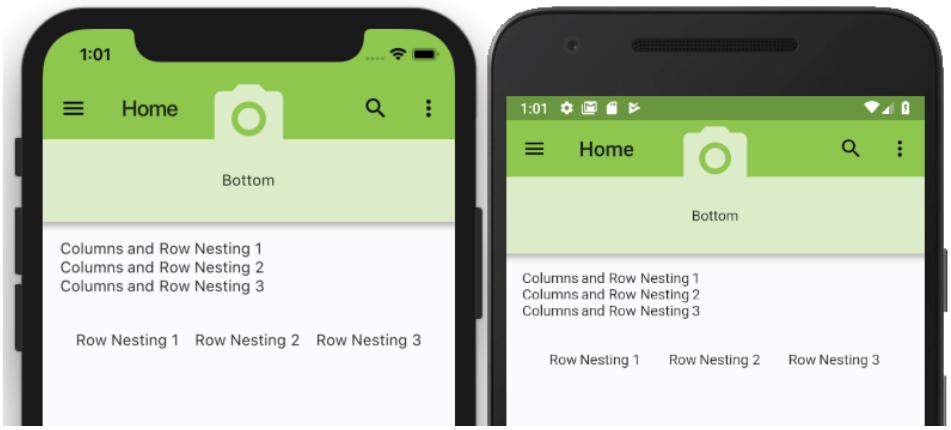


FIGURE 6.7: Column and Row widgets rendered in app

TRY IT OUT Adding Column, Row, and Nesting the Row and Column together as Widget Classes

You'll add three widget classes to the `body` property section of the `Column` list of widgets. Between each widget class, you'll add a simple `Divider()` widget to draw separation lines between sections.

1. Add the widget class names `ColumnWidget()`, `RowWidget()`, and `ColumnAndRowNestingWidget()` to the `Column` children widget list. The `Column` widget is located in the `body` property. Add a `Divider()` widget between each widget class name. Make sure each widget class uses the `const` keyword.

```
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. Create the `ColumnWidget()` widget class after the `ContainerWithBoxDecorationWidget()` widget class.

```
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. Create the `RowWidget()` widget class after the `ColumnWidget()` widget class.

```
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. Create the `ColumnAndRowNestingWidget()` widget class after the `RowWidget()` widget class.

```
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'),
          ],
        ),
      ],
    );
  }
}
```

How It Works

`Column` and `Row` are handy widgets to lay out either vertically or horizontally. Nesting the `Column` and `Row` widgets creates flexible layouts needed for each circumstance. Nesting widgets is at the heart of designing Flutter UI layouts.

Buttons

There are a variety of buttons to choose from, depending on the situation, such as `FloatingActionButton`, `FlatButton`, `IconButton`, `RaisedButton`, `PopupMenuButton`, and `ButtonBar`.

FloatingActionButton

The `FloatingActionButton` widget is usually placed on the bottom right or center of the main screen in the `Scaffold` `floatingActionButton` property. Use the `FloatingActionButtonLocation` widget to either dock (notch) or float above the navigation bar. To dock a button to the navigation

bar, use the `BottomAppBar` widget. By default, it's a circular button but can be customized to a stadium shape by using the named constructor `FloatingActionButton.extended`. In the example code, I commented out the stadium shape button for you to test.

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

Figure 6.8 shows the `FloatingActionButton` widget on the bottom right of the screen with the notch enabled.

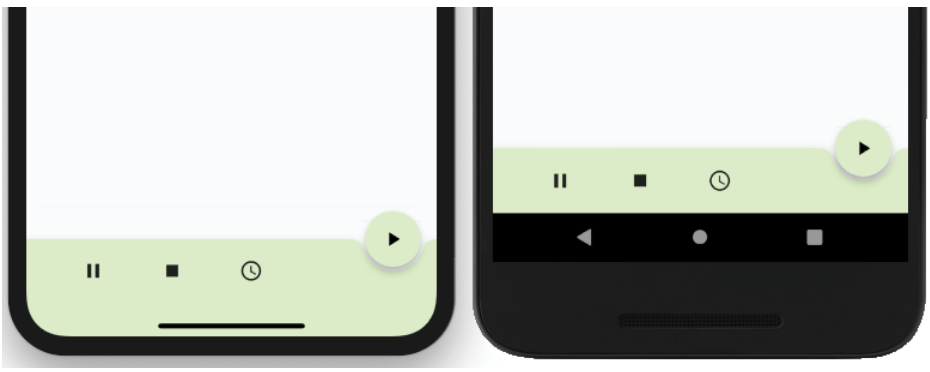


FIGURE 6.8: FloatingActionButton with notch

FlatButton

The `FlatButton` widget is the most minimalist button used; it displays a text label without any borders or elevation (shadow). Since the text label is a widget, you could use an `Icon` widget instead or another widget to customize the button. `color`, `highlightColor`, `splashColor`, `textColor`, and other properties can be customized.

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

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

Figure 6.9 shows the default `FlatButton` widget on the left and the customized `FlatButton` widget on the right.

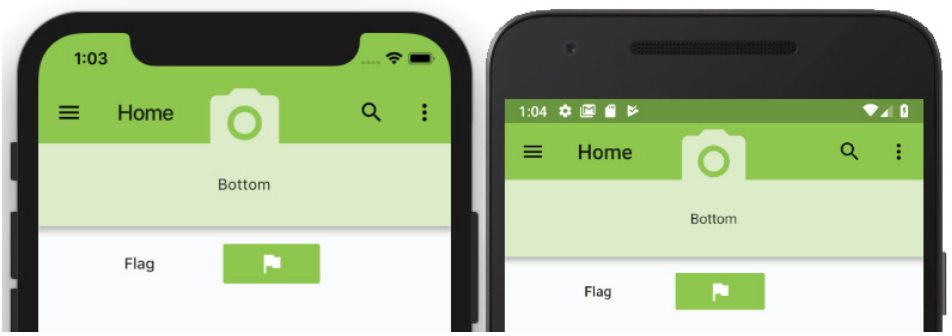


FIGURE 6.9: FlatButton

RaisedButton

The `RaisedButton` widget adds a dimension, and the elevation (shadow) increases when the user presses the button.

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

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

Figure 6.10 shows the default `RaisedButton` widget on the left and the customized `RaisedButton` widget on the right.

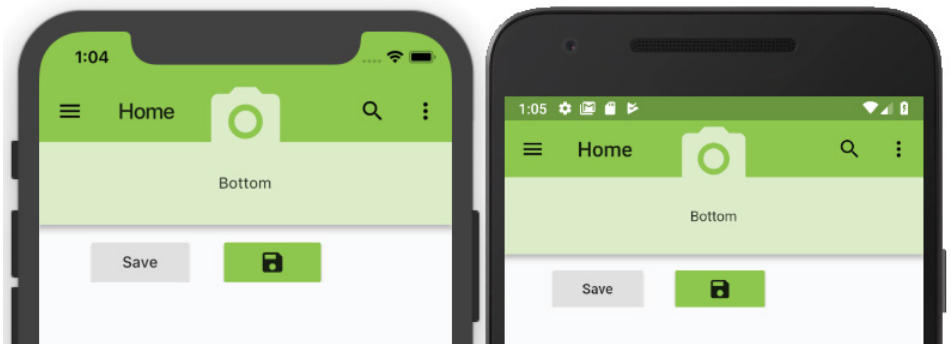


FIGURE 6.10: RaisedButton

IconButton

The `IconButton` widget uses an `Icon` widget on a Material Component widget that reacts to touches by filling with color (ink). The combination creates a nice tap effect, giving the user feedback that an action has started.

```
// 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',
),
```

Figure 6.11 shows the default `IconButton` widget on the left and the customized `IconButton` widget on the right.

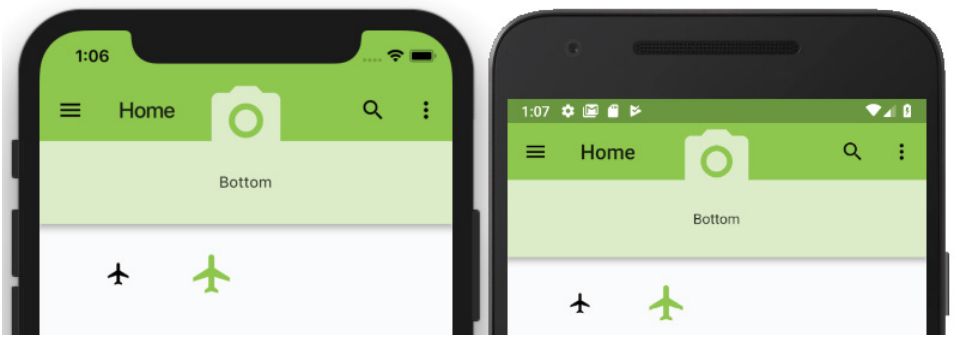


FIGURE 6.11: IconButton

PopupMenuButton

The `PopupMenuButton` widget displays a list of menu items. When a menu item is pressed, the value passes to the `onSelected` property. A common use of this widget is placing it on the top right of the `AppBar` widget for the user to select different menu options. Another example is to place the `PopupMenuButton` widget in the middle of the `AppBar` widget showing a list of search filters.

TRY IT OUT Creating the PopupMenuButton and the Items' Class and List

Before you add the `PopupMenuButton` widgets, let's create the `Class` and `List` necessary to build the items to be displayed. Usually, the `TodoMenuItem` (model) class would be created in a separate Dart file, but to keep the example focused, you'll add it to the `home.dart` file. In the final three chapters of this book, you'll separate classes into their own files.

1. Create a `TodoMenuItem` class. When you create this class, make sure it's not inside another class. Create the class and list at the end of the file after the last closing curly bracket, }. The `TodoMenuItem` class contains a `title` and an `icon`.

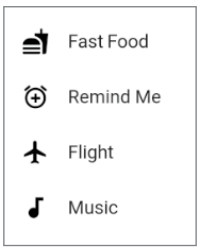
```
class TodoMenuItem {
  final String title;
  final Icon icon;

  TodoMenuItem({this.title, this.icon});
}
```

2. Create a `List` of `TodoMenuItem`. This `List<TodoMenuItem>` will be called `foodMenuList` and will contain a `List` (array) of `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. Create a `PopupMenuButton`. You will use an `itemBuilder` to build the `List` of `TodoMenuItems`. If you do not set an icon for the `PopupMenuButton`, a default menu icon is used by default. The `onSelected` will retrieve the item selected on the list. Use the `itemBuilder` to build a list of `foodMenuList` and map to `TodoMenuItem`. A `PopupMenuItem` is returned for each item in the `foodMenuList`. For the `PopupMenuItem` child, you use a `Row` widget to show the `Icon` and `Text` widgets together.

```
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. Modify the `AppBar` `bottom` property by adding the widget class name: `PopupMenuButtonWidget()`.

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

5. Create the `PopupMenuButtonWidget()` widget class after the `ColumnAndRowNestingWidget()` widget class. Since the `bottom` property is expecting a `PreferredSizeWidget`, you use the keyword `implements PreferredSizeWidget` in the class declaration. The class extends the `StatelessWidget` and implements the `PreferredSizeWidget`.

After the widget build, implement the `@override` `preferredSize` getter; this is a required step because the purpose of `PreferredSizeWidget` is to provide the size for the widget; in this example, you'll set the `height` property. Without this step, we'd have no size specified.

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

The following is the entire `PopupMenuButtonWidget` widget class. Note that the `Container` widget's `height` property uses the `preferredSize.height` property that you set in the `PreferredSize-Widget` 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);
}
```

How It Works

The `PopupMenuButton` widget is a great widget to display a `List` of items such as menu choices. For the list of items, you created a `TodoMenuItem Class` to hold a `title` and `icon`. You created the

`foodMenuList`, which is a `List` of each `TodoMenuItem`. In this case, the `List` items are hard-coded, but in a real-world app, the values can be read from a web service. In Chapters 14, 15 and 16, you'll implement Cloud Firestore to access data from a web server.

ButtonBar

The `ButtonBar` widget (Figure 6.12) aligns buttons horizontally. In this example, the `ButtonBar` widget is a child of a `Container` widget to give it a background color.

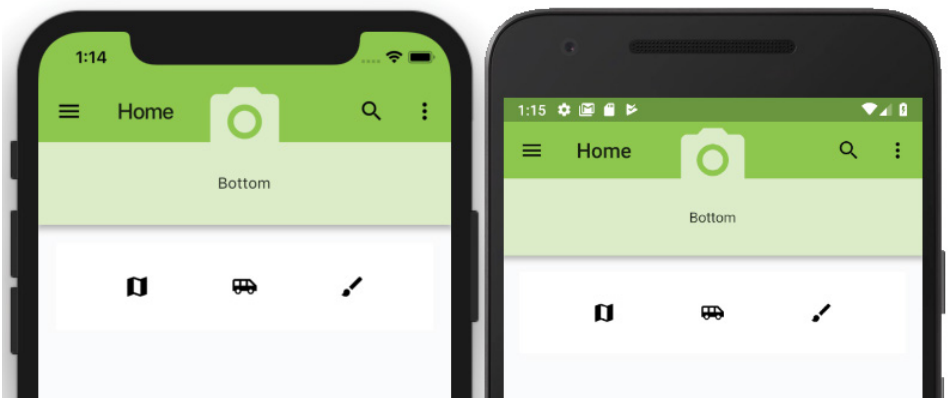


FIGURE 6.12: ButtonBar

```
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: () {},
      ),
    ],
  ),
),
```

TRY IT OUT Adding Buttons as Widget Classes

You've looked at the `FloatingActionButton`, `FlatButton`, `RaisedButton`, `IconButton`, `PopupMenuButton`, and `ButtonBar` widgets. Here you'll create two widget classes to organize the buttons' layout.

1. Add the widget class names `ButtonsWidget()` and `ButtonBarWidget()` to the `Column` children widget list. The `Column` is located in the `body` property. Add a `Divider()` widget between each widget class name. Make sure each widget class uses the `const` keyword.

```
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. Create the `ButtonsWidget()` widget class after the `ColumnAndRowNestingWidget()` widget class. The class returns a `Column` with three `Row` widgets for the `children` list of `Widget`. Each `Row` `children` list of `Widget` contains different buttons such as the `FlatButton`, `RaisedButton`, and `IconButton` buttons.

```
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: () {},
    ),
  ],
),
```

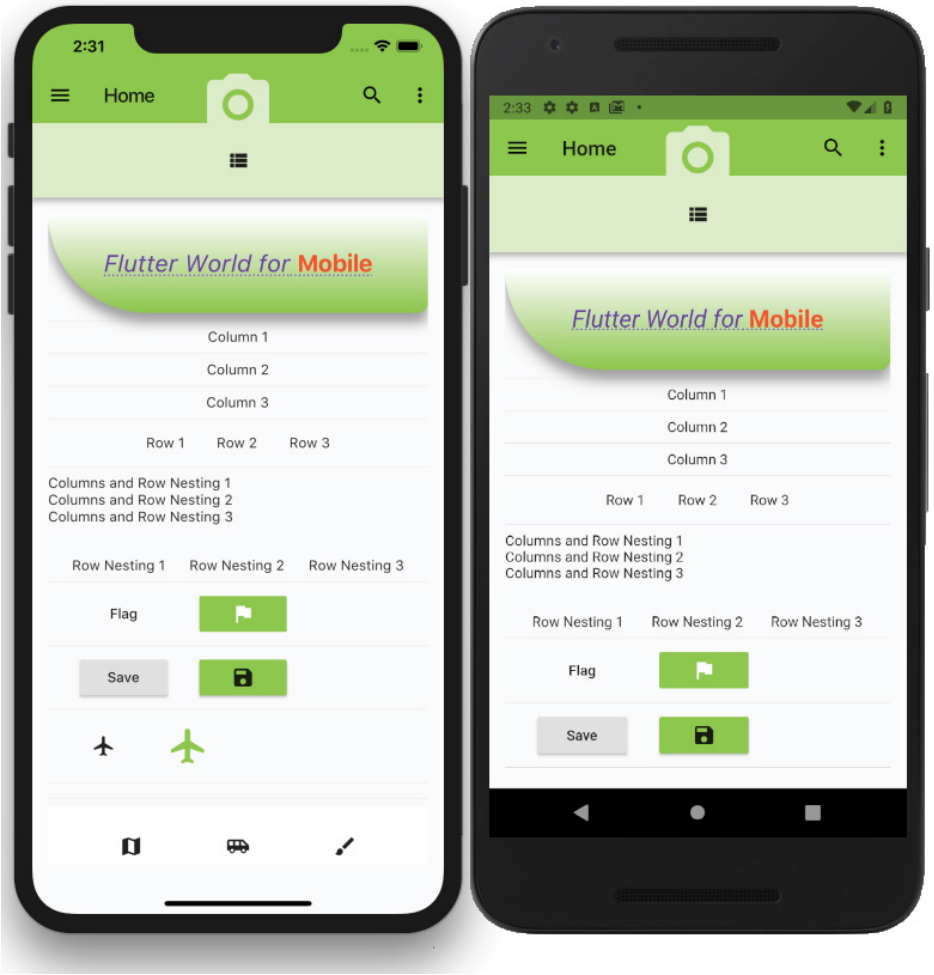
```
        Divider(),
      ],
    );
  }
}
```

3. Create the `ButtonBarWidget()` widget class after the `ButtonsWidget()` widget class. The class returns a `Container` with a `ButtonBar` as a `child`. The `ButtonBar children` list of `Widget` contains three `IconButton` widgets.

```
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: () {},
          ),
        ],
      ),
    );
  }
}
```

How It Works

The `FloatingActionButton`, `FlatButton`, `RaisedButton`, `IconButton`, `PopupMenuButton`, and `ButtonBar` widgets are configurable by setting the properties `icon`, `iconSize`, `tooltip`, `color`, `text`, and more.



USING IMAGES AND ICONS

Images can make an app look tremendous or ugly depending on the quality of the artwork. Images, icons, and other resources are commonly embedded in an app.

AssetBundle

The `AssetBundle` class provides access to custom resources such as images, fonts, audio, data files, and more. Before a Flutter app can use a resource, you must declare it in the `pubspec.yaml` file.

```
// 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
```

Instead of declaring each asset, which can get very long, you can declare all the assets in each directory. Make sure you end the directory name with a forward slash, `/`. Throughout the book, I'll use this approach when adding assets to the projects.

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

Image

The `Image` widget displays an image from a local or URL (web) source. To load an `Image` widget, there are a few different constructors to use.

- `Image()`—Retrieves image from an `ImageProvider` class
- `Image.asset()`—Retrieves image from an `AssetBundle` class using a key
- `Image.file()`—Retrieves image from a `File` class
- `Image.memory()`—Retrieves image from a `Uint8List` class
- `Image.network()`—Retrieves image from a `URL` path

Press Ctrl+Spacebar to invoke the code completion for the available options (Figure 6.13).

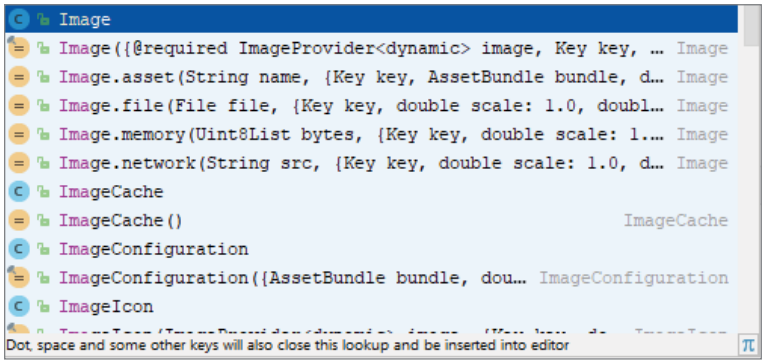


FIGURE 6.13: Image code completion

As a side note, the `Image` widget also supports animated GIFs.

The following sample uses the default `Image` constructor to initialize the `image` and `fit` arguments. The `image` argument is set by using the `AssetImage()` constructor with the default bundle location of the `logo.png` file. You can use the `fit` argument to size the `Image` widget with the `BoxFit` options, such as `contain`, `cover`, `fill`, `fitHeight`, `fitWidth`, or `none` (Figure 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',
),
```

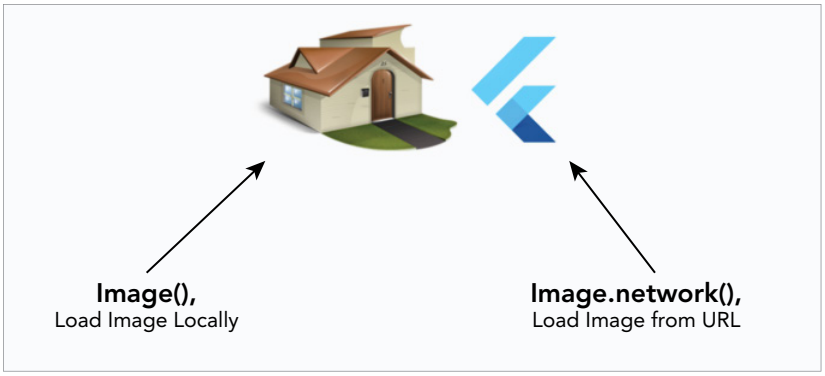


FIGURE 6.14: Images loaded locally and from network (web)

If you add color to the image, it colorizes the image portion and leaves any transparencies alone, giving a silhouette look (Figure 6.15).

```
// Image
  Image(
  image: AssetImage("assets/images/logo.png"),
  color: Colors.deepOrange,
  fit: BoxFit.cover,
),
```

FIGURE 6.15: Silhouette-style image

Icon

The `Icon` widget is drawn with a glyph from a font described in `IconData`. Flutter's `icons.dart` file has the full list of icons available from the font `MaterialIcons`. A great way to add custom icons is to add to the `AssetBundle` fonts containing glyphs. Once example is Font Awesome, which has a high-quality list of icons and a Flutter package. Of course, there are many other high-quality icons available from other sources.

The `Icon` widget allows you to change the `Icon` widget's `color`, `size`, and other properties (Figure 6.16).

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

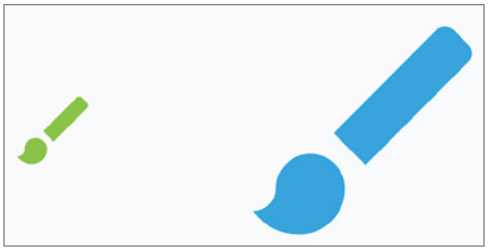


FIGURE 6.16: Icons with custom sizes

TRY IT OUT Creating the Images Project; Adding Assets; and Loading Images, Icons, and Decorators

Create a new Flutter project and name it `ch6_images`; you can follow the instructions in Chapter 4. For this project, you need to create only the `pages` and `assets/images` folders. Create the `Home` class as a `StatelessWidget`. The goal of this app is to provide a look at how to use the `Image` and `Icon` widgets.

In this example, you'll customize the `width` property of the two `Image` widgets according to the device screen size. To obtain the device screen size, you can use the `MediaQuery.of()` method.

1. Open the `pubspec.yaml` file to add resources. In the `assets` section, add the `assets/images/` folder declaration. I like to create an `assets` folder at the root of the project and add subfolders for each type of resource, as shown in Chapter 4.

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

   Add the folder `assets` and subfolder `images` at the project's root and then copy the `logo.png` file to the `images` folder. Click the Save button, and depending on the editor you are using, it automatically runs `flutter packages get`. Once finished, it shows this message: `Process finished with exit code 0`. If it does not automatically run the command for you, open the Terminal window (located at the bottom of your editor) and type **`flutter packages get`**.

2. Open the `home.dart` file and modify the `body` property. Add a `SafeArea` widget to the `body` property with a `SingleChildScrollView` as a `child` of the `SafeArea` widget. Add `Padding` as a `child` of `SingleChildScrollView` and then add a `Column` as a `child` of the `Padding`.

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

3. Add the widget class name `ImagesAndIconWidget()` to the `Column` children widget list. The `Column` is located in the `body` property.

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

4. Add the `ImagesAndIconWidget()` widget class after `class Home extends StatelessWidget {...}`. In the widget class, a local image is loaded by the `AssetImage` class. Using the `Image.network` constructor an image is loaded by a URL string. The `Image` widget's `width` property uses the `MediaQuery.of(context).size.width / 3` to calculate the `width` value as one-third of the device width.

```
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,
        ),
      ],
    );
  }
}
```

How It Works

By declaring your assets in the `pubspec.yaml` file, they are accessible by the `AssetImage` class from an `AssetBundle`. The `Image` widget through the `image` property loads a local image with the `AssetBundle` class. To load an image over a network (such as the Web), you use the `Image.network` constructor by passing a URL string. The `Icon` widget uses the `MaterialIcons` font library, which draws a glyph from the font described in the `IconData` class.

USING DECORATORS

Decorators help to convey a message depending on the user's action or customize the look and feel of a widget. There are different types of decorators for each task.

- `Decoration`—The base class to define other decorations.
- `BoxDecoration`—Provides many ways to draw a box with `border`, `body`, and `boxShadow`.
- `InputDecoration`—Used in `TextField` and `TextFormField` to customize the border, label, icon, and styles. This is a great way to give the user feedback on data entry, specifying a hint, an error, an alert icon, and more.

A `BoxDecoration` class (Figure 6.17) is a great way to customize a `Container` widget to create shapes by setting the `borderRadius`, `color`, `gradient`, and `boxShadow` properties.

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

FIGURE 6.17: BoxDecoration applied to a Container

The `InputDecoration` class (Figure 6.18) is used with a `TextField` or `TextFormField` to specify labels, borders, icons, hints, errors, and styles. This is helpful in communicating with the user as they enter data. For the `border` property shown here, I am implementing two ways to customize it, with `UnderlineInputBorder` and with `OutlineInputBorder`:

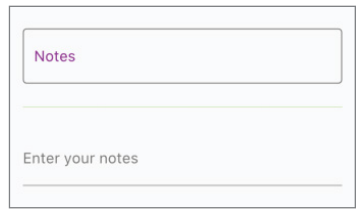


FIGURE 6.18: InputDecoration with OutlineInputBorder and default border

```
// 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',
    ),
  ),
```

TRY IT OUT Continuing the Images Project by Adding Decorators

Still editing the `home.dart` file, you'll add the `BoxDecoratorWidget()` and `InputDecoratorsWidget()` widget classes.

1. Add the widget class names `BoxDecoratorWidget()` and `InputDecoratorsWidget()` after the `ImagesAndIconWidget()` widget class. Add a `Divider()` widget between each widget class name.

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

2. Add the `BoxDecoratorWidget()` widget class after the `ImagesAndIconWidget()` widget class. The widget class returns a `Padding` widget with the `Container` widget as a child. The `Container` `decoration` property uses the `BoxDecoration` class. Using the `BoxDecoration` `borderRadius`, `color`, and `boxShadow` properties, you create a rounded button shape such as the one in Figure 6.17.

```
class BoxDecoratorWidget extends StatelessWidget {
  const BoxDecoratorWidget({
```

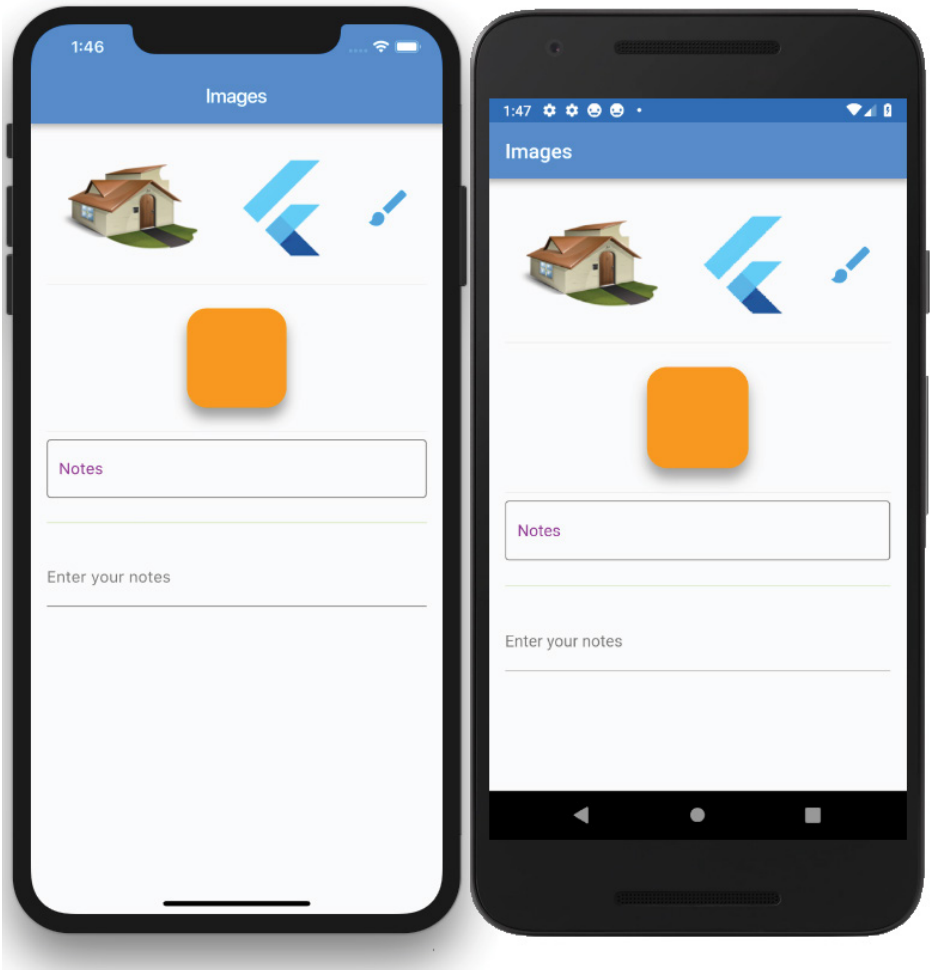
```
    Key key,
  }) : super(key: key);

  @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. Add the `InputDecoratorsWidget()` widget class after the `BoxDecoratorWidget()` widget class. You take a `TextField` and use `TextStyle` to change the `color` and `fontSize` properties. The `InputDecoration` class is used to set the `labelText`, `labelStyle`, `border`, and `enabledBorder` values to customize the border properties. I am using the `OutlineInputBorder` here, but you could also use the `UnderlineInputBorder` class instead. I left `border` `UnderlineInputBorder` and `enabledBorder` `OutlineInputBorder()` commented out, allowing you to test both classes.

   The following code adds two `TextField` widgets customized by two different decorations. The first `TextField` customizes different `InputDecoration` properties to show a purple notes label with the `OutlineInputBorder()`. The second `TextField` widget uses the decoration without customizing the border property.

```
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'),
        ),
      ],
    );
  }
}
```



How It Works

Decorators are invaluable to enhance the look and feel of widgets. The `BoxDecoration` provides many ways to draw a box with `border`, `body`, and `boxShadow`. The `InputDecoration` is used in either a `TextField` or `TextFormField`. Not only does it allow the customization of the `border`, `label`, `icon`, and `styles`, but it also gives users feedback on data entry with hints, errors, icons, and more.

USING THE FORM WIDGET TO VALIDATE TEXT FIELDS

There are different ways to use text field widgets to retrieve, validate, and manipulate data. The `Form` widget is optional, but the benefits of using a `Form` widget are to validate each text field as a group. You can group `TextFormField` widgets to manually or automatically validate them. The `TextFormField` widget wraps a `TextField` widget to provide validation when enclosed in a `Form` widget.

If all text fields pass the `FormState validate` method, then it returns `true`. If any text fields contain errors, it displays the appropriate error message for each text field, and the `FormState validate` method returns `false`. This process gives you the ability to use `FormState` to check for any validation errors instead of checking each text field for errors and not allowing the posting of invalid data.

The `Form` widget needs a unique key to identify it and is created by using `GlobalKey`. This `GlobalKey` value is unique across the entire app.

In the next example, you'll create a form with two `TextFormFields` (Figure 6.19) to enter an item and quantity to order. You'll create an `Order` class to hold the item and quantity and fill the order once the validation passes.

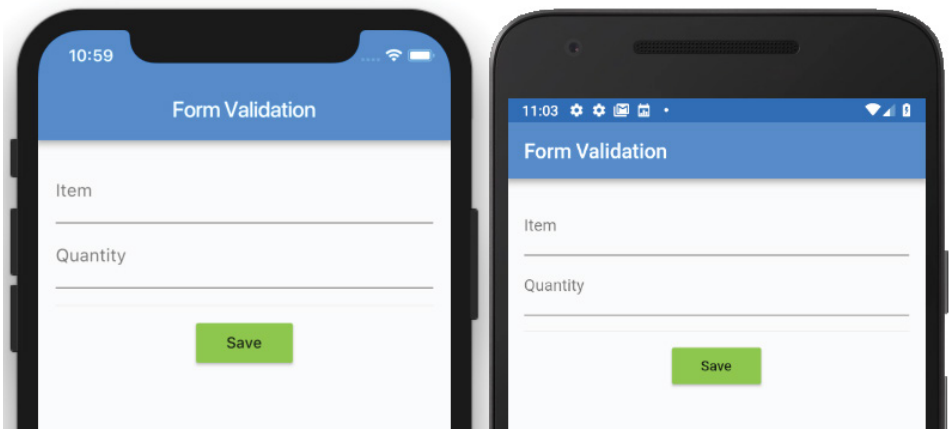


FIGURE 6.19: The Form and TextFormField layout

TRY IT OUT Creating the Form Validation App

Create a new Flutter project and name it `ch6_form_validation`. You can follow the instructions in Chapter 4. For this project, you need to create only the `pages` folder. The goal of this app is to show how to validate data entry values.

1. Open the `home.dart` file and add to the body a `SafeArea` widget with `Column` as a `child`. In the `Column` children, add the `Form()` widget, which you modify in step 7.

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

2. Create the `Order` class after `class _HomeState extends State<Home> {...}`. The `Order` class will hold `item` as a `String` value and `quantity` as an `int` value.

```
class _HomeState extends State<Home> {
  //...
}

class Order {
  String item;
  int quantity;
}
```

3. After the `class _HomeState extends State<Home>` declaration and before `@override`, add the variables `_formStateKey` for the `GlobalKey` value and `_order` to initiate the `Order` class.

   Create the unique key for the form by using `GlobalKey<FormState>` and mark it `final` since it will not change.

```
class _HomeState extends State<Home> {
  final GlobalKey<FormState> _formStateKey = GlobalKey<FormState>();

  // Order to Save
  Order _order = Order();
```

4. Create the `_validateItemRequired(String value)` method that accepts a `String` value. Use the ternary operator to check whether the value is set to `isEmpty`, and if yes, then return `'Item Required'`. Otherwise, return `null`.

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

5. Create the `_validateItemCount(String value)` method that accepts a `String` value. Use the ternary operator to convert `String` to `int`. Then check whether `int` is greater than zero; if it's not, return `'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. Create the `_submitOrder()` method called by the `FlatButton` widget to check whether all `TextFormField` fields pass validation and call `Form save()` to gather values from all `TextFormFields` to the `Order` class.

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

7. Add to the `Form()` widget a private key variable called `_formStateKey`, set `autovalidate` to `true`, add `Padding` for the `child` property, and add `Column` as a child of `Padding`.

   Setting `autovalidate` to `true` allows the `Form()` widget to check validation for all fields as the user enters information and to display an appropriate message. If `autovalidate` is set to `false`, no validation happens until the `_formStateKey.currentState.validate()` method is manually called.

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

      ],
    ),
  ),
),
```

8. Add two `TextFormField` widgets to the `Column children` list of `Widget`. The first `TextFormField` is an item description, and the second `TextFormField` is a quantity of items to order.
9. Add an `InputDecoration` class with `hintText` and `labelText` for each `TextFormField`.

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

10. Add a call for the `validator` and `onSaved` methods. The `validator` method is called to validate characters as they are entered, and the `onSaved` method is called by the `Form save()` method to gather values from each `TextFormField`.

    For the validator, pass the value entered in the `TextFormField` widget by naming the variable `value` inside parentheses and use the fat arrow syntax (`=>`) to call the method `_validateItemRequired(value)`. The fat arrow syntax is shorthand for `{ return mycustomexpression; }`.

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

Note that in step 2 you created an `Order` class to hold the item and quantity values to be collected by the `onSaved` methods. When the `Form save()` method is called, all of the `TextFieldForm onSaved` methods are called, and values are collected in the `Order` class such as `order.item = value`.

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

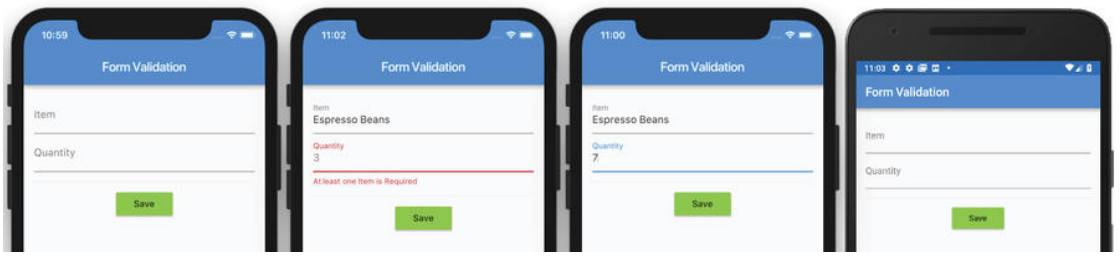
The following code shows both `TextFormFields`:

```
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),
),
```

Notice that you use `int.tryParse()` to convert the quantity value from `String` to `int`.

11. Add a `Divider` and a `RaisedButton` after the last `TextFormField`. For the `onPressed`, call the `_submitOrder()` method created in step 6.

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



How It Works

When retrieving data from input fields, the `Form` widget is an incredible helper, and you used the `GlobalKey` class to assign a unique key to identify it. Use the `Form` widget to group `TextFormField` widgets to manually or automatically validate them. The `FormState validate` method validates data, and if it passes, it returns `true`. If the `FormState validate` method fails, it returns `false`, and each text field displays the appropriate error message. Each `TextFormField` validator property has a method to check for the appropriate value. Each `TextFormField onSaved` property passes the currently entered value to the `Order` class. In a real-world app, you would take the `Order` class values and save them to a database locally or on a web server. In Chapters 14, 15 and 16, you'll learn how to implement Cloud Firestore to access data from a web server.

CHECKING ORIENTATION

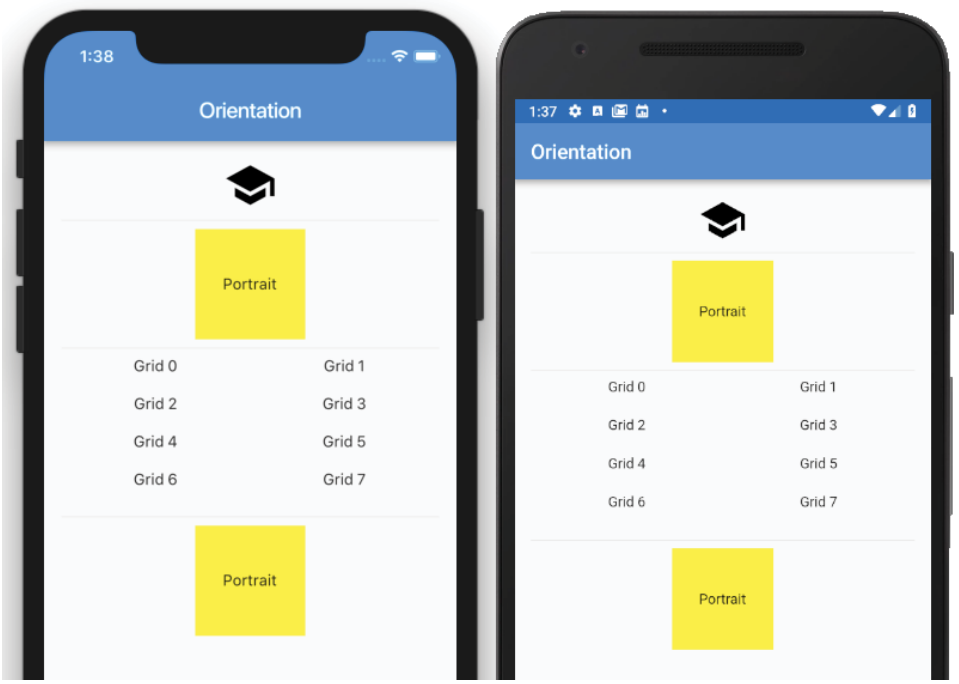
Under certain scenarios, knowing the device orientation helps in laying out the appropriate UI. There are two ways to figure out orientation, `MediaQuery.of(context).orientation` and `OrientationBuilder`.

A huge note on `OrientationBuilder`: it returns the amount of space available to the parent to figure out orientation. This means it does not guarantee the actual device orientation. I prefer using `MediaQuery` to obtain the actual device orientation because of its accuracy.

TRY IT OUT Creating the Orientation App

Create a new Flutter project and name it `ch6_orientation`. You can follow the instructions in Chapter 4. For this project, you only need to create the `pages` folder.

In this example, the UI layout will change depending on orientation. When the device is in portrait mode, it will show one `Icon`, and when in landscape mode, it will show two `Icons`. You'll take a look at a `Container` widget that will grow in size and change color, and you'll use a `GridView` widget to show two or four columns. Lastly, I added the `OrientationBuilder` widget to show that when the `OrientationBuilder` is not a parent widget, the correct orientation is not calculated correctly. But if you place the `OrientationBuilder` as a parent, it works correctly; note that using `SafeArea` does not affect the outcome. The following image shows the final project.



1. Open the `home.dart` file and add to the body a `SafeArea` with `SingleChildScrollView` as a `child`. Add `Padding` as a child of the `SingleChildScrollView`. Add a `Column` as a `child` of the `Padding`. In the `Column children` property, add the widget class called `OrientationLayoutIconsWidget()`, which you will create next. Make sure you add the `const` keyword before the widget class name to take advantage of caching to improve performance.

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

2. Add the `OrientationLayoutIconsWidget()` widget class after `class Home extends StatelessWidget {...}`. The first variable to initialize is the current orientation by calling `MediaQuery.of()` after `Widget build(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();
  }
}
```

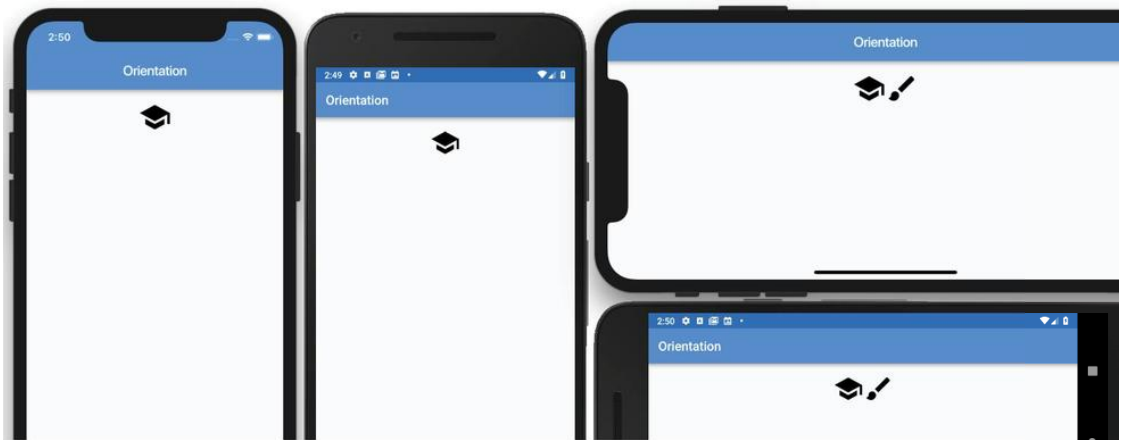
3. Based on the current `Orientation`, you return a different layout of `Icon` widgets. Use a ternary operator to check whether `Orientation` is `portrait`, and if so, return a single `Row` icon. If `Orientation` is `landscape`, return a `Row` of two `Icon` widgets. Replace the current `return Container()` with the following code:

```
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. Putting all of the code together, you get the following:

```
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. After `OrientationLayoutIconsWidget()`, add a `Divider` widget and the `OrientationLayoutWidget()` widget class to create.

   The steps are similar to the earlier ones, but instead of using rows and icons, you are using containers: obtain the `Orientation` mode and for portrait return a yellow `Container` widget with a width of 100.0 pixels. When the device is rotated, the landscape returns a green `Container` widget with a width of 200.0 pixels.

```
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. After `OrientationLayoutWidget()`, add a `Divider` widget and the `GridViewWidget()` widget class that you will create.

Although you will take a closer look at the `GridView` widget in Chapter 9, it is appropriate to use it now since it's the closest to a real-world example. In portrait mode, the `GridView` widget shows two columns, and in landscape mode, it shows four columns.

There are a few items to note here. Since the `GridView` widget is inside a `Column` widget, set the `GridView.count constructor shrinkWrap` argument to `true` or it will break the constraints. I also set the `physics` argument to `NeverScrollableScrollPhysics()` or the `GridView` will scroll its children from within. Remember, you have all these widgets inside a `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. After `GridViewWidget()`, add a `Divider` widget and the `OrientationBuilderWidget()` widget class that you will create.

As mentioned previously, I use `MediaQuery.of()` to obtain orientation because it's more accurate, but it's good to know how to use `OrientationBuilder`.

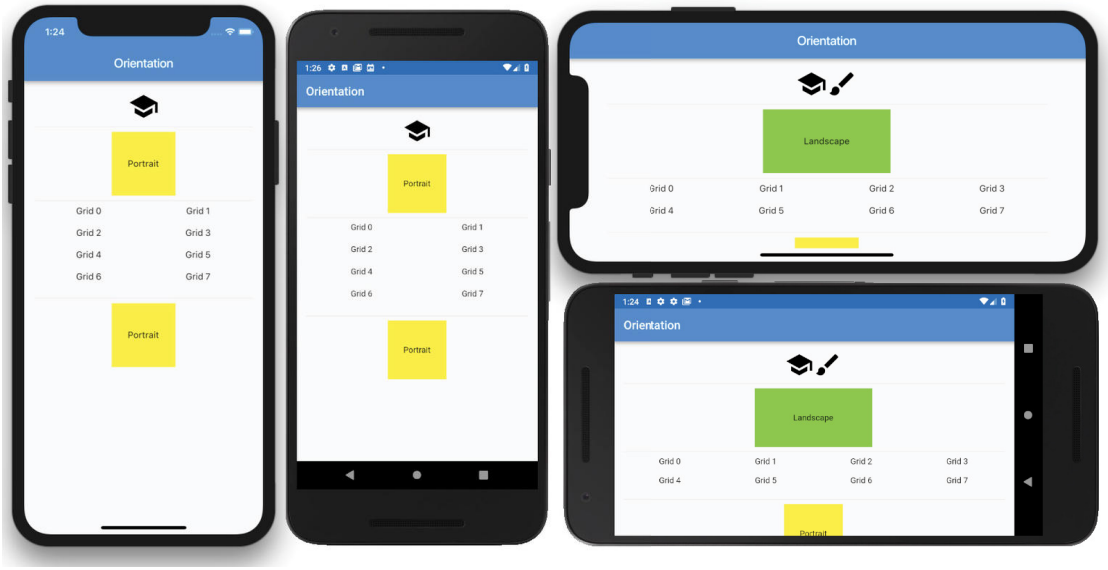
`OrientationBuilder` requires a `builder` property to be passed and cannot be `null`. The `builder` property takes two parameters: `BuildContext` and `Orientation`.

```
builder: (BuildContext context, Orientation orientation) {}
```

The steps and result are the same as with `_buildOrientationLayout()`. Use the ternary operator to check for the orientation, and for portrait, and return a yellow `Container` widget with a width of 100.0 pixels. When the device is rotated, the landscape returns a green `Container` widget with a width of 200.0 pixels.

Note that `OrientationBuilder` runs the risk of not detecting the orientation mode correctly because it is a `child` widget and relies on the parent screen size instead of the device orientation. Because of this, I recommend using `MediaQuery.of()` instead.

```
// 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'),
        );
      },
    );
  }
}
```



How It Works

You can detect device orientation by calling `MediaQuery.of(context).orientation`, which returns either a `portrait` or `landscape` value. There is also `OrientationBuilder`, which returns the amount of space available to the parent to figure out the orientation. I recommend using `MediaQuery` to retrieve the correct device orientation.

SUMMARY

In this chapter, you learned about the most commonly used (basic) widgets. These basic widgets are the building blocks to designing mobile apps. You also explored different types of buttons to choose depending on the situation. You learned how to add assets to your app via `AssetBundle` by listing items in the `pubspec.yaml` file. You used the `Image` widget to load images from the local device or a web server through a URL string. You saw how the `Icon` widget gives you the ability to load icons by using the MaterialIcons font library.

To modify the appearance of widgets, you learned how to use `BoxDecoration`. To improve giving users feedback on data entry, you implemented `InputDecoration`. Validating multiple text field data entries can be cumbersome, but you can use the `Form` widget to manually or automatically validate them. Lastly, using `MediaQuery` to find out the current device orientation is extremely powerful in any mobile app to lay out widgets depending on the orientation.

In the next chapter, you'll learn how to use animations. You'll start by using widgets such as `AnimatedContainer`, `AnimatedCrossFade`, and `AnimatedOpacity` and finish with the powerful `AnimationController` for custom animation.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Using basic widgets	You learned to use `Scaffold`, `SafeArea`, `AppBar`, `Container`, `Text`, `RichText`, `Column`, `Row`, `Column and Row Nesting`, `Buttons`, `FloatingActionButton`, `FlatButton`, `RaisedButton`, `IconButton`, `PopupMenuButton`, and `ButtonBar`.
Using images	You learned to use `AssetBundle`, `Image`, and `Icon`.
Using decorators	You learned to use `Decoration`, `BoxDecoration`, and `InputDecoration`.
Using forms for text field validation	You learned to use the `Form` widget to validate each `TextFormField` as a group.
Detecting orientation	You learned to use `MediaQuery.of(context).orientation` and `OrientationBuilder` to detect device orientation.